

06/09/00  
jc841 U.S. PTO

06-12-00

A

PTO/SB/05 (12/97)

Approved for use through 09/30/00. OMB 0651-0032  
Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number

<b>UTILITY PATENT APPLICATION TRANSMITTAL</b> (Only for new nonprovisional applications under 37 CFR 1.53(b))		Attorney Docket No.: TD-155	
		First Named Inventor: Baldwin	
		Title: Graphic Memory Management With Invisible Hardware-Managed Page Faulting	
		Express Mail Label No. EL519157967US	
<b>APPLICATION ELEMENTS</b> See MPEP Chapter 600 concerning utility patent application contents		ADDRESS TO: Assistant Commissioner for Patents Box Patent Application Washington, DC 20231	
1. <u>    </u> Fee Transmittal Form (e.g., PTO/SB/17) (Submit an original, and a duplicate for fee processing)		5. <u>    </u> Incorporation by Reference (useable if Box 4b is checked) The entire disclosure of the prior application, from which a copy of the oath or declaration is supplied under Box 4b, is considered to be part of the disclosure of the accompanying application and is hereby incorporated by reference therein.	
2. <u>✓</u> Specification Total pages <u>1+77</u> (preferred arrangement set forth below) <ul style="list-style-type: none"><li>- Descriptive title of the Invention</li><li>- Cross References to Related Applications</li><li>- Statement Regarding Fed sponsored R&amp;D</li><li>- Reference to Microfiche Appendix</li><li>- Background of the Invention</li><li>- Brief Summary of the Invention</li><li>- Brief Description of the Drawings (if filed)</li><li>- Detailed Description</li><li>- Claim(s)</li><li>- Abstract of the Disclosure</li></ul>		6. <u>    </u> Microfiche Computer Program (Appendix)	
3. <u>✓</u> Drawing(s) (35USC d113) Total pages <u>17</u>		7. <u>    </u> Nucleotide and/or Amino Acid Sequence Submission (if applicable, all necessary) <ul style="list-style-type: none"><li>a. <u>    </u> Computer Readable Copy</li><li>b. <u>    </u> Paper Copy (identical to computer copy)</li><li>c. <u>    </u> Statement verifying identity of above copies</li></ul>	
4. Oath of Declaration Total pages <u>    </u> <ul style="list-style-type: none"><li>a. <u>    </u> Newly Executed (original or copy)</li><li>b. <u>    </u> Copy from a prior application (37 CFR 1.63(d)) (for continuation/divisional with Box 17 completed)<ul style="list-style-type: none"><li>i. <u>    </u> Deletion of inventor(s) (Signed statement attached deleting inventor(s) named in the prior application, see 37, C.F.R. 1.63(d)(2) and 1.33(b).)</li></ul></li></ul>		8. <u>    </u> Assignment Papers (cover sheet & Documents(s)) 9. <u>    </u> 37 CFR §3.73(b) Statement (when there is an assignee) <u>    </u> Power of Attorney 10. <u>    </u> English Translation Document (if applicable) 11. <u>    </u> Information Disclosure Statement (IDS)/PTO-1449 <u>    </u> Copies of IDS Citations 12. <u>    </u> Preliminary Amendment 13. <u>✓</u> Return Receipt Postcard (MPEP 503) (Should be specifically itemized) 14. <u>    </u> *Small Entity Statement(s) (PTO/SB/09-12) <u>    </u> Statement filed in prior application -Status still proper and desired 15. <u>    </u> Certified Copy of Priority Document(s) . 16. <u>    </u> Other:  * A new statement is required to be entitled to pay small entity fees, except where one has been filed in a prior application and is being relied upon.	
17. If a CONTINUING APPLICATION, check appropriate box and supply the requisite information below and in a preliminary amendment: <u>  x  </u> Continuation <u>    </u> Divisional <u>    </u> Continuation-in-part (CIP) of prior application No: 60/138,350 filed 06/09/99, 60/138,248 filed 06/09/99 and 60/143,822 filed 7/13/99.  Prior application information: Examiner: Group / Art Unit:			
18. CORRESPONDENCE ADDRESS: Name: Groover & Associates p.c. Address: 17000 Preston Road, Suite 230, Dallas, Texas 75248 (Dallas County) Phone: (972) 380-6333, fax (972) 380-4445			
Signature <u>Robert Groover</u> Robert Groover, Reg.No.30,039, Date: June 8, 2000			

---

# Graphics Memory Management With Invisible Hardware-Managed Page Faulting

---

U.S. Patent Application of:

David R. Baldwin,

Inventor

3Dlabs Inc., Ltd.,

Assignee

Attorney's Docket No. TD-155

Robert Groover, REG.PAT.ATTY.

000051-521650

# Graphics Memory Management

## With Invisible Hardware-Managed Page Faulting

### Background and Summary of the Invention

The present application relates to computer graphics rendering systems and methods,  
5 and particularly to handling of texture data used by rendering accelerators for 3D graphics.

#### Background: 3D Computer Graphics

One of the driving features in the performance of most single-user computers is  
computer graphics. This is particularly important in computer games and workstations, but  
is generally very important across the personal computer market.

10 For some years the most critical area of graphics development has been in  
three-dimensional ("3D") graphics. The peculiar demands of 3D graphics are driven by the  
need to present a realistic view, on a computer monitor, of a three-dimensional scene. The  
pattern written onto the two-dimensional screen must therefore be derived from the three-  
dimensional geometries in such a way that the user can easily "see" the three-dimensional  
15 scene (as if the screen were merely a window into a real three-dimensional scene). This  
requires extensive computation to obtain the correct image for display, taking account of  
surface textures, lighting, shadowing, and other characteristics.

The starting point (for the aspects of computer graphics considered in the present  
application) is a three-dimensional scene, with specified viewpoint and lighting (etc.). The  
20 elements of a 3D scene are normally defined by sets of polygons (typically triangles), each  
having attributes such as color, reflectivity, and spatial location. (For example, a walking  
human, at a given instant, might be translated into a few hundred triangles which map out  
the surface of the human's body.) Textures are "applied" onto the polygons, to provide detail  
in the scene. (For example, a flat carpeted floor will look far more realistic if a simple  
25 repeating texture pattern is applied onto it.) Designers use specialized modelling software  
tools, such as 3D Studio, to build textured polygonal models.

The 3D graphics pipeline consists of two major stages, or subsystems, referred to as  
geometry and rendering. The **geometry** stage is responsible for managing all polygon  
activities and for converting three-dimensional spatial data into a two-dimensional  
30 representation of the viewed scene, with properly-transformed polygons. The polygons in  
the three-dimensional scene, with their applied textures, must then be transformed to obtain  
their correct appearance from the viewpoint of the moment; this transformation requires  
calculation of lighting (and apparent brightness), foreshortening, obstruction, etc.

The most challenging 3D graphics applications are dynamic rather than static. In addition to changing objects in the scene, many applications also seek to convey an illusion of movement by changing the scene in response to the user's input. Whenever a change in the orientation or position of the camera is desired, every object in a scene must be recalculated relative to the new view. As can be imagined, a fast-paced game needing to maintain a high frame rate will require many calculations and many memory accesses.

**Figure 2** shows a high-level overview of the processes performed in the overall 3D graphics pipeline. However, this is a very general overview, which ignores the crucial issues of what hardware performs which operations.

## Hardware Acceleration

Since rendering is a computationally intensive operation, numerous designs have offloaded it from the main CPU. An example of this is the GLINT chip described below.

20 Texturing

There are different ways to add complexity to a 3D scene. Creating more and more detailed models, consisting of a greater number of polygons, is one way to add visual interest to a scene. However, adding polygons necessitates paying the price of having to manipulate more geometry. 3D systems have what is known as a "polygon budget," an approximate number of polygons that can be manipulated without unacceptable performance degradation. In general, fewer polygons yield higher frame rates.

The visual appeal of computer graphics rendering is greatly enhanced by the use of "textures." A texture is a two-dimensional image which is mapped into the data to be rendered. Textures provide a very efficient way to generate the level of minor surface detail which makes synthetic images realistic, without requiring transfer of immense amounts of data. Texture patterns provide realistic detail at the sub-polygon level, so the higher-level tasks of polygon-processing are not overloaded. See Foley et al., *Computer Graphics:*

A typical graphics system reads data from a texture map, processes it, and writes color data to display memory. The processing may include mipmap filtering which requires access to several maps. The texture map need not be limited to colors, but can hold other information that can be applied to a surface to affect its appearance; this could include height perturbation to give the effect of roughness. The individual elements of a texture map are called "texels."

Awkward side-effects of texture mapping occur unless the renderer can apply texture maps with correct perspective. Perspective-corrected texture mapping involves an algorithm that translates "texels" (pixels from the bitmap texture image) into display pixels in accordance with the spatial orientation of the surface. Since the surfaces are transformed (by the host or geometry engine) to produce a 2D view, the textures will need to be similarly transformed by a linear transform (normally projective or "affine"). (In conventional terminology, the coordinates of the object surface, i.e. the primitive being rendered, are referred to as an (s,t) coordinate space, and the map of the stored texture is referred to a (u,v) coordinate space.) The transformation in the resulting mapping means that a horizontal line in the (x,y) display space is very likely to correspond to a slanted line in the (u,v) space of the texture map, and hence many additional reads will occur, due to the texturing operation, as rendering walks along a horizontal line of pixels.

## Data and Memory Management

Due to the extremely high data rates required at the end of the rendering pipeline, many features of computer architecture take on new complexities in the context of computer graphics (and especially in the area of texture management).

30 Virtual Memory Management

One of the basic tools of computer architecture is "virtual" memory. This is a

technique which allows application software to use a very large range of memory addresses, without knowing how much physical memory is actually present on the computer, nor how the virtual addresses correspond to the physical addresses which are actually used to address the physical memory chips (or other memory devices) over a bus.

5        Some further discussion of Virtual memory management can be found in Hennessy  
& Patterson, Computer Architecture: a Quantitative Approach (2.ed.1996); Hwang and  
Briggs, Computer Architecture and Parallel Processing (1984); Subieta, Object-based virtual  
memory for PCs (1990); Carr, Virtual memory management (1984); Lau, Performance  
improvement of virtual memory systems (1982); and Loshin, Efficient Memory Programming  
10 (1998); all of which are hereby incorporated by reference. An excellent hypertext tutorial  
is found in the Web pages which start at <http://cne.gmu.edu/Modules/VM/>, and this hypertext  
tutorial is also hereby incorporated by reference. Another useful online resource is found at  
<http://www.harlequin.com/mm/reference/faq.html>, and this too is hereby incorporated by  
reference. Much current work can be found in the annual proceedings of the ACM  
15 International Symposium on Memory Management (ISMM), which are all hereby  
incorporated by reference.

## AGP and GART

Beginning with the Pentium II<sup>■</sup>, some Intel processors have included the capability for an Accelerated Graphics Port (AGP). The AGP provides a high-speed dedicated bus for fast transfer of graphics data. (Unlike the PCI bus, the AGP bus is pipelined, and allows only two devices on it.)

To support this high-speed bus, the Intel specification also provides a special protocol for "AGP memory." This is not physically separate memory, but just dynamically-allocated system DRAM areas which the graphics chip can access quickly. The Intel chip set includes address translation hardware which makes the "AGP memory" look continuous to the graphics controller. This permits the graphics chip to access large texture bitmaps (e.g. 128KB) as a single entity.

Intel's built-in chip set hardware is called the GART (Graphics Address Remapping Table). The GART hardware is somewhat similar in function to the paging hardware in the CPU chip, in that the processor "linear" virtual addresses get automatically translated into physical addresses (which may point to system RAM and local Frame Buffer memory, as well as the AGP RAM).

However, this translation is fairly inflexible, and completely out of the user's control.

Thus it cannot be optimized for particular applications, software architectures, or graphics accelerator architectures.

### Image Copying and Scaling

One common operation in computer graphics is to copy a rectangular image to the screen, but only draw certain parts of it. For example, a texture image may be stored on an otherwise blank page; when the texture image is desired to be inserted into a display, the blank background page is obviously unneeded. The parts of the source image not to be copied are defined by setting them to a specific color, called the "key" color. During the copy, a test is made for the existence of this key color, and any pixels of this key color are rejected and therefore not copied. This technique allows an image of any shape to be copied onto a background, since the unwanted pixels are automatically excluded. For example, this could be used to show an explosion, where the flames are represented by an image.

As the explosion continues, or as the viewer moves closer to it, its size increases. This effect is produced by scaling the image during the copy. Magnifying the image produces unwanted side effects, however, and the final image may appear blocky and unconvincing. An example of this technique is shown in Figures 1A and 1B. In these figures, the gray area represents the desired image, and the black area represents the key color. Figure 1A shows the original texture, and Figure 1B shows the same image copied and scaled. Note that the unwanted key color area has been removed cleanly, but the staircase effect on the edge is magnified. When a texture has more than one color on the interior of the object, as is usually the case, the interior of the scaled texture will also be blocky and unattractive, since there will be no smooth transition between blocks of different color.

The normal way to deal with this is to bilinear-filter the image during the copy so that pixels in the source image are blended with their neighbors to remove the blocky effect. As described above, this procedure blends the color of a given pixel with the colors of that pixel's nearest neighbors, to produce a smoother image overall. This works within the valid parts of the image, but leaves extremely blocky edges. Figures 1C and 1D show an original texture, and same texture after it has been filtered, copied, and scaled, respectively. Note that in this case, the cut out edge is as blocky as the original example, but in addition the edge pixels have the black (key color) background blended with the correct color, giving a dark border.

There are three primary artifacts, or defects in the resulting image, caused by bilinear





is disk storage.

The swapping required for virtual memory management is normally done automatically (as far as the application software is concerned). There is a vast amount of literature concerning CPU based virtual memory systems and their management.

5 The apparently-larger virtual texture memory space increases performance as the optimum set of textures (or part of textures) are chosen for residence by the hardware. It also simplifies the management of texture memory by the driver and/or application where either or both try to manage the memory manually. This is akin to program overlays before the days of virtual memory on CPUs where the program had to dynamically load and unload  
10 segments of itself.

The present inventor has realized that managing the texture memory in the driver or by the application is very difficult (or impossible) to do properly, because:

1. What does the driver/application do when it runs out of memory and needs to fit another texture in? Which texture(s) does it delete?
- 15 2. The texture has to be completely resident and physically contiguous so a large enough space must be made available.
3. A texture which is about to be used MUST NOT be deleted or moved: otherwise all command buffers will be outdated.
4. In some cases a texture map will not fit into memory even when all other textures are  
20 deleted (a 2Kx2K 32bpp texture map takes 16MBytes of memory).
5. The texture heap must be compacted to reclaim storage.

The idea of applying virtual management techniques to textures in 3D graphics hardware appears to be suggested, for example, by U.S. Patent 5,790,130 to Gannett. This patent suggests that "A graphics hardware device, coupled to the host computer, renders  
25 texture mapped images, and includes a local memory that stores at least a portion of the texture data stored in the system memory at any one time. A software daemon runs on the processor of the host computer and manages transferring texture data from the system memory to the local memory when needed by the hardware device to render an image." (Abstract) This and/or other virtual texture memory schemes are believed to have been used  
30 in some products of HP and SGI. However, the present inventor has realized that these schemes are ill-suited for most personal computer applications (and many workstation applications). The main aim in these implementation seems to have been to allow very large texture maps (16Mx16M or larger) to be used. By contrast, the innovations in the present

application are not motivated only by desire for such large maps, but to remove the software problems in managing the comparatively small amount of texture storage (vs the large amounts of texture storage in SGI and HP machines) efficiently. Thus it is possible that the architectural innovations disclosed herein can be used in combination with those used by SGI and HP.

### **Graphics Memory Management With Invisible Hardware-Managed Page Faulting**

As noted above, virtual memory architectures have long been used in general-purpose computers. However, there turn out to be some surprising difficulties in using this idea in computer graphics (especially for texture memory). The present application discloses several innovations related to virtualization and caching of texture memory.

In particular, the present application discloses a computer system in which a graphics accelerator unit manages page faulting of texture data invisibly to the host processor. When a logical page fault occurs and the page of texture is in the second level of memory (i.e. the host's physical memory) it will be fetched in automatically by the graphics memory manager, and the host is not aware anything has happened. For this to happen a number of automatic mechanisms must be in place:

- a. Determine where the page is located in host physical memory.
- b. Determine which page out of the working set (in level 1 memory) to use. In a sample embodiment, this determination uses the least recently used algorithm.
- c. Make this page the most recently used page (as well as continuing to keep the least-recently-used list up to date as other pages are used).
- d. Update the page tables for the new page and remove any reference to the page just bumped out of memory (if any).
- e. Download the page.
- f. Restart texture processing.

Note that if the faulting logical page identifies a page in the third level memory the host does (a) (after having made the page available), but the hardware carries on and does b, c, d, e and f.

It should be noted that, once an interrupt is issued to get memory services, what happens in hardware is not a concern for the host nor for the rendering software.

Notable (and separately innovative) features of the virtual texture mapping architecture described in the present application include at least the following: A single chip solution is provided; Two or three levels of texture memory hierarchy are supported; The page faulting







disclosed embodiments of the read-modify-write solutions in its rendering subsystem. A sample board incorporating the P3™ graphics processor may include these elements:

- the P3™ graphics core itself;
- a PCI/AGP interface;
- DMA controllers for PCI/AGP interface to the graphics core and memory;
- SGRAM/SDRAM, to which the chip has read-write access through its frame buffer (FB) and local buffer (LB) ports;
- a RAMDAC, which provides analog color values in accordance with the color values read out from the SGRAM/SDRAM; and
- a video stream interface for output and display connectivity.

Various claimed features, and/or features of particular interest, are found in the Texture Read Unit, which will now be described in detail.

#### **Texture Read Unit Description**

The Texture Read Unit's main job is to manage the primary texture cache (the data part is in the Texture Filter Unit) and load texel data into it, preferably in advance of when it is needed. The primary cache can be used as one large cache or as two smaller (half size) caches depending on the type of texture mapping being done. The single large cache is an optimization and allows higher cache hits when the texture map is large or the polygon is large and a single bilinear texture is used.

When texture needs to be loaded the address(es) are calculated for the texel data. These addresses may be physical addresses in which case the address is issued to the Memory Controller and some time later the data returned. Alternatively the address may be a logical one so the following steps are taken to resolve (or translate) it:

- The logical address (really just the page part) is looked up in the Translation Look aside Buffer (TLB) and if present the corresponding physical address is issued to the Memory Controller.
- The address translation may fail in the TLB so the page table in memory is accessed and if the page is resident the physical address is looked up, the TLB updated and the physical address is issued to the Memory Controller.
- The page may not be resident in the working set so the page is read from host memory (or the host asked for it via an interrupt) and when it has been loaded the newly updated page table is read, the TLB updated and the physical address is issued to the Memory Controller. The page may be marked as a host texture in which case



[illegible][illegible][illegible][illegible]

- [illegible]

[illegible][illegible]

- [illegible]



It is the software's responsibility to set the layout to Patch32\_2 or Patch2 as appropriate when the texture map is downloaded. The hardware will write the texel data into the correct place but not switch layouts automatically.

The minimum width and height of a texture map (in any layout) is 2 texels. If the width and/or height of a texture map is 1 (such as the lowest resolution map in a set of mip maps) then the texels must be replicated to expand the offending dimension(s) to 2 texels. (This is necessary as the bilinear index calculations will select set  $i0 = 0$  and  $i1 = 0$  with the interpolation coefficient set to combine some fraction of texel 0 with texel 0, thus yielding texel 0. The Filter Unit cannot be configured to do this, but can achieve the same effect if the texel is repeated.) If a 1x1 texture map has a border then the 3x3 map is stored as a 4x4 map as shown:

b0	b1	b2
b3	b4	b5
b6	b7	b8

b0	b1	b2	b2
b0	b1	b2	b2
b3	t0	b4	b4
b5	b6	b7	b7

All the preceding comments on texture map layout for 1D and 2D texture maps apply to the individual slices of a 3D texture map. The base address of slice 0 is given in the TextureBaseAddr0 register and the offset (in texels) between slices is given in TextureMapSize register. The TextureMapSize should be set to a value greater than or equal

to the product of the width and height for a slice.

### Address Calculation

The type of texture is checked and if it is a 3D texture map the base address is set from TextureBaseAddr[0] register, the layout and texel size are taken from TextureReadMode0 register and the width from TextureMapWidth0.

If the texture is not a 3D texture map then the layout, texel size and width parameters are taken from the appropriate texture registers (these registers should be loaded the same for per pixel mip mapping). The width is divided by 2 to the (map level), so the correct mip map width is used. Note the width does not have to be a power of 2, so the divide may have a remainder (which is ignored) so will fail past some map level. This is not a problem as mip maps will always be a power of two in size and non mip maps will always have a map level of 0. The base address is read from one of the 16 base address registers. The actual one used depends on the map level, the map base level and map max level associated with this texture as given by:

offset into base registers = min(texture map level + map base level, max map level)

so the allocation of the base registers between the two possible textures is up to software.

The maximum width is 4095, but the minimum width depends on the layout as the Patch2 and Patch32\_2 have some minimum requirements. If the mip mapping forces the width below these minimum requirements then the width is forced to be the minimum allowed for the texel size. The minimum texel widths are 8, 4 and 2 for 8, 16 and 32 bits per texel respectively. The minimum width is one memory word (i.e. 16 bytes). Also if the width falls below 128, 64 or 32 texels for 8, 16 or 32 bits per texel respectively any textures with a Patch32\_2 layout are automatically set to Patch2.

The address is calculated as follows. (i and j are the coordinates of the required texel.)

**For linear layout** the pixel offset is:

bottom left origin:  $-j * \text{width} + i$

top left origin:  $j * \text{width} + i$ .

**For Patch64** the 2D ij coordinate space is mapped to a 1D address range as shown in Figure 5, in which Pixel Offset (top left origin) is given by:

```
i % 64 +          // i within a patch
(i / 64) * 1024 +  // i between patches
(j % 16) * 64 +    // j within a patch
```

```
(j / 16) * width * 16 // j between patches
```

This can be converted into a simpler calculation just using shifts and adds:

```
(i & 0x3f) + ((i & 0xffc0) << 4) + ((j & 0xf) << 6) + ((j & 0xfff0) * width).
```

For bottom left origin the equation is:

```
(i & 0x3f) + ((i & 0xffc0) << 4) - ((j & 0xf) << 6) - ((j & 0xfff0) * width)
```

**For Patch2** the 2D ij coordinate space is mapped to a 1D address range as shown in the following equations:

Pixel Offset (top left origin) is given by:

```
i % 2 + // i within a patch
(i / 2) * 4 + // i between patches
(j % 2) * 2 + // j within a patch
(j / 2) * width * 2 // j between patches
```

This can be converted into a simpler calculation just using shifts and adds (only top left origin is supported):

```
(i & 0x1) + ((i & 0xffff) << 1) + ((j & 0x1) << 1) + ((j & 0xffff) * width)
```

**For Patch32\_2** the 2D ij coordinate space is mapped to a 1D address range as shown in the following equations:

First calculate the offset to the corresponding 2x2 patch (recall there are 16x16 within a 1K page):

```
i' = i >> 1
j' = j >> 1
(i' % 16 + // i within a 32 x 32 patch
(i' / 16) * 256 + // i between 32 x 32 patches
(j' % 16) * 16 + // j within a 32 x 32 patch
(j' / 16) * width * 8) * 4 + // j between 32 x 32 patches
// convert from 2x2 patches to texels
```

Add in the offset within the 2x2 sub patch:

```
i % 2 + // i within a patch
(j % 2) * 2 // j within a patch.
```

This can be converted into a simpler calculation just using shifts and adds (only top left origin is supported):

```
((i' & 0xf) + ((i' & 0xffff) << 4) + ((j' & 0xf) << 4) +
((j' & 0xffff) * width) >> 1) << 2) +
(i & 0x1) + ((j & 0x1) << 1).
```

For a 3D texture the TextureMapSize (in texels) is multiplied by the k index (which selects the slice) to get the offset to the start of the slice the texel is on:

texelOffset += k \* TextureMapSize.

Note that the TextureMapSize does not have to be width x height, but can be larger, if necessary.

Convert the texel offset into a byte offset, based on the texel size:

```
8bpp:    byteOffset = texelOffset * 1
```

```
16bpp:    byteOffset = texelOffset * 2
```

32bpp:  $\text{byteOffset} = \text{texelOffset} * 4$

64bpp:  $\text{byteOffset} = \text{texelOffset} * 8$

Add in the base address for the texture map. The base address is held as a byte address and must be aligned to the natural boundary for the texel size. For a 16bpp address the bottom bit must be 0. For a 32 bpp address the bottom two bits must be zero. This is forced in hardware to remove any concerns of what happens if this condition is not true.

```
8bpp:    byteAddr = baseAddr + byteOffset
```

```
16bpp:  byteAddr = (baseAddr & ~0x1) + byteOffset
```

```
32bpp:    byteAddr = (baseAddr & ~0x3) + byteOffset
```

```
64bpp:    byteAddr = (baseAddr & ~0x7) + byteOffset
```

All address calculations are done to 32 bit and any overflow just wraps around. The  $i$  and  $j$  coordinates are zero extended up to the required width. The bottom 4 bits of the texel's byte address give the start byte in the memory's 128 bit width and the remaining upper bits give the memory address.

## Primary Cache

An efficient texture cache is vital if a sustained texture rate of one output texel per cycle is to be achieved and maintained. This is even more important when mip mapping as, in general, the zoom ratio is between 1:1 and 2:1 (output:input) so there is only moderate re-use of texel data as we move from one pixel to the next.

One way to improve this is to try to hold enough texels in the cache so that some re-use of them can be made on the next scanline. If this can be done then only one new texel per output pixel on the second scanline is needed for bilinear filtering, otherwise 2 new texels are needed. For mip mapping this translates to 1.5 new texels when making use of scanline coherence or 3 new texels without. These figures can be improved on by organizing the texel data in memory more efficiently and this will be covered later once the organizational details have been covered.

Clearly the best scheme is when all the texture map fits into cache however, at this

point in time, this is not feasible except for the smallest of texture maps (32x32 at 16 bits per texel).

The cache is divided into two banks so two independent textures can be cached without any interference, or to hold two levels of a mip map, or slices of a 3D texture. When a single non mip mapped texture is being used the two caches can be joined together so a larger texture map or polygon can be rendered while still maintaining scanline coherency.

Span processing where the pixel mask (as part of the SpanStep message) is modified by the texel data does not use the primary cache.

The cache is always enabled and the only control over its operation the user has is to be able to invalidate the cache. This needs to be done whenever a new texture map is selected or the current texture map's data is edited in memory, thus causing any cached data become stale.

The cache is divided into two parts: a data part and a directory part.

## Data Part

The data part holds the texel data and this can be found in the Texture Filter Unit so it is connected directly to the linear interpolators used to implement the filtering operations.

The texel data is held in "raw" format so the cache holds the maximum number of texels and the texel data is converted "on the fly" as it is needed into 8888 format the filter logic expects. The two texel formats which cannot be handled this way is the 8 bit indexed textures (replicating the conversion LUT is too expensive) and YUV 422 (the addressing and data routing gets too complicated). In these two cases the data is converted into 8888 formats and this is loaded into the cache.

Each cache line holds 128 bits of data and there are 256 cache lines in each bank for RX and 64 cache lines in each bank for P3. (These sizes are for illustration only and may be changed later.) Each cache line holds a 2x2, 4x2 or 8x2 patch of texels for 32, 16 and 8 bits per texel respectively. In the 2x2 case the cache's performance is independent of the traversal direction through the texture map, however in the other two cases the "u" direction is preferred over the "v" direction.

The patch (2x2, etc.) has a fixed relationship to the origin of the texture map such that the origin of the patch is always some integral multiple of the patch size from the origin of the texture map. The following diagram shows the 2x2 patch arrangement within a texture map. The numbers in the brackets show the texel coordinates within the texture map vary and the T0...T3 are the corresponding filter registers each texel is assigned.

The grey areas are show the texels held in a memory word (16 bytes) for each size of texel. The texture map may also be patched at a higher level (32x32) to reduce the effect of page breaks but this is of no consequence to how the primary cache functions (see **Figure 6**).

The organization of texture maps within memory is important and tries to meet several criteria:

- The performance should be independent of the traversal direction, especially for "large" texture maps (i.e.  $> 32 \times 32$ ). Storing the texture map in a linear fashion gives very good access times in the u direction but poor access times in the v direction due to the page organization of DRAMS. Storing the texture maps in a patch form (32x32 in our case for 32 bit texels) equalizes the access times.
- The memory width is very much wider than the texel width so each memory read returns multiple texels. If the texel data in a memory word are all for the same row then all the data is used when traversing in u (along a row) but very little is used in the v direction (along a column). The 2x2 patch organization ensures that at least 2 texels can be used from each memory read for all traversal directions.

Texture maps are preferably stored in memory in one of the 2x2 patched formats to give the best overall performance for general 3D use, however this is not always possible or desirable. For example if the texture data originates from an external source or is used to drive an external device (i.e. a monitor) the layout of the data may be fixed and not in 2x2 format. Alternatively the traversal direction may be known to always be in the u direction - examples of this are video scaling, fonts and general 2D use.

When the texture map is stored in memory in a non 2x2 layout it is formatted into the 2x2 layout expected by the Filter Unit as it is read in.

The layout in memory for the various supported format is shown in **Figures 7A-7B**. Each line is one memory word and the bit numbers are shown along the top. The tick marks are at byte intervals and the numbers in brackets show how the texel coordinates vary within the memory word.

Note in the Linear and Patch64 cases only one alignment has been shown. The origin can be in 4, 8 or 16 places with respect to the width of the memory word (16 bytes) for 32, 16 or 8 bit texels respectively.

### **Directory Part**

The directory part of the primary cache is held in this unit and is searched to find out if a texel is already in the primary cache, and if so where. The search is done fully associatively and 8 texels (four per cache bank) are searched simultaneously (to support the target performance of trilinear filtering or two bilinear filtered texels in a cycle). The replacement policy is oldest first (FIFO). These parameters will be justified later.

The key stored in the cache directory is formed from the texel's integer coordinate (i, j) and map level (or k for 3D texture). A bank of the cache cannot hold texels from different texture maps (texels from the different levels in a mip map or from the different slices in a 3D texture can be held in the same bank). This means that the cache *must* be invalidated whenever a new texture map is selected.

Why not use the texel's address as the key then the cache can hold texels from different maps and does not need to be invalidated when a different texture map is selected? The answer is that the address calculation for 8 texels would need to be done in parallel and this would be quite expensive. This unit is supplied i0, i1 and j0, j1 indices (these would be necessary for the address computation) and the four texels (just considering one bank) are given by (i0, j0), (i1, j0), (i0, j1) and (i1, j1).

The typical search policies are fully associative, set associative and direct mapped. These are graded from most expensive, most flexible (fully associative) to least expensive, least flexible (direct mapped). Set associative and direct mapped both rely on using a subset of index bits to choose one (direct mapped) or a set of locations to search.

The access patterns through a 2D texture map follow an approximate straight line. (It is actually a slightly curved line due to the perspective projection, but this is a minor effect and doesn't change any of the reasoning.) The orientation of the line and its position is arbitrary and successive scanline will all follow on approximately parallel paths. The other variable to contend with is the width of the texture map - this is variable (between texture maps) and a power of two. Given these constraints choosing a set of index bits to which will give a good distribution for each possible orientation of line looks an impossible task. A good distribution is vital otherwise, in the worst case, all texels along a line could fall into one set (or a single entry for direct mapped) - clearly this will defeat the purpose of a cache. The fully associative search works equally well for all access patterns.

The common replacement policies are least recently used (LRU), oldest (FIFO), least frequently used and random. The LRU policy usually gives excellent result but is the

most expensive, however the approximately regular access patterns repeated from scanline to scanline will make the least recently used page the same as the oldest page (at least within the same polygon). The oldest replacement policy is implemented by a simple counter which selects the entry to replace and is incremented after every replacement. The counter wraps within the available table size.

An alternative replacement policy (KeepOldest) is also supported which is the same as oldest up until the point the cache is about to wrap on a scanline (i.e. earlier cache lines on the scanline are going to be replaced). When the wrap occurs rather than starting back at the first entry used on the scanline the last  $n$  entries are reused as scratch cache lines. This prevents scanline coherency from being lost for the whole of the next scanline, but places some restrictions on the amount of expedited loading of the cache which can be done. It is difficult to know how big the scratch area should be for the best trade off between maintaining some scanline coherency and more stalling due to less effective cache loads. The value of  $n$  is programmable (the TextureCacheReplacementMode).

The size of the cache is a compromise - the larger the better, but it follows the law of diminishing returns. The minimum useful size is based on the number of texels visited along any path through the texture map. This will be the minimum of the texture map size and width of the polygon. The cache is patched based so holds a minimum of two rows (maybe only partial rows) at a time. The filter may require texels from two adjacent patches (in  $v$ ) so in the worst case two pairs of rows are needed. If a bank holds  $n$  bytes of data the maximum width of texture map (or texels along a polygon) which can be held while maintaining scanline coherency is  $n / (\text{bytes per texel}) / 4$ .

For P3 each bank has 1K bytes of storage so for 16 bit textures the cache works best when less than 128 texels are used for mip maps or 256 texels for a single texture map (where both caches can be combined).

For RX each bank has 4K bytes of storage so for 32 bit textures the cache works best when less than 256 texels are used for mip maps or 512 texels for a single texture map (where both caches can be combined).

The fully associative search is expensive and the two factors which govern the cost are the number of entries to search and the width of the key. The number of entries is governed by the cache line length and the total amount of data in the cache bank. The cache line length and size of the cache have already been considered, but what about the key?

The key (as already described) holds the  $i$  and  $j$  index and the map level (3D textures



will be considered shortly). The maximum width and height of a map is 2050 ( $2K + a$  border) so the indices have 12 bits. The cache line holds a  $2 \times 2$  patch so the indices can be reduced by one bit to 11 bits. The number of map level is needed here. In total the key is  $(11 + 11 + 4)$  bits or 26. This can be reduced down to 23 by realizing that the full  $2050 \times 2050$  value can only occur on map level 0. Map level 1 has a maximum size of  $1026 \times 1026$  so by encoding the map into the upper bits as shown in **Figure 8**, the key width can be reduced.

report that a texel is present and this is used to select which register file is to supply the texel data. This bank select bit is passed to the Filter Unit in the T4BorderColor to T7BorderColor bits as these are not needed in this mode of operation.

## Loading the Cache

Any caching scheme is going to suffer from cache misses where the only option open is to go and read the texel data from memory. The latency for the data to return may be anything from a few cycles to many tens of cycles depending on how busy the memory is and if the texture request introduces a page break. (This assumes that the texture is resident in memory or is a physical texture. If the texture is non-resident then the time for it to be fetched from host memory could be thousands of cycles at best or many more if the host has to respond to an interrupt, page the texture off disk and then download it.)

A fragment could cause from one to eight memory reads, although if the cache is working well and scanline coherency is being made use of this will very much reduced. (The pathological case is where bilinear filtering is being done with a zoom ratio of  $1:n$ , where  $n > 1$ . In this case we are minifying the map and no coherence between adjacent fragments or scanlines can be exploited. From 1 to 4 reads per fragment are needed depending on how the sample points interact with the underlying  $2 \times 2$  patch structure in the texture map.) **Figure 9** shows which texels the memory reads bring in and the corresponding output fragments they will satisfy. The zoom ratio of 1:1 is used as this is the worst case for mip mapping and occurs for the higher resolution map; the lower resolution map will have a zoom ratio of 2:1 so any results for this map level will be twice as good. A texel size of 32 bits is also assumed so these results are independent of any path orientation. The smaller texels sizes will give better results for X major paths.

From this figure it can be seen that for the interior fragments on even rows no memory reads are required (because all the texel data was read in for the previous odd row) while for odd rows there is a read for every other fragment, F(number of reads):

Even rows:  $F(0), F(0), F(0), F(0), F(0), F(0), F(0), \text{etc.}$

Odd rows:  $F(1), F(0), F(1), F(0), F(1), F(0), F(1), \text{etc.}$

The next lower resolution map:

Row 0:  $F(0)$ ,  $F(0)$ ,  $F(0)$ ,  $F(0)$ ,  $F(0)$ ,  $F(0)$ ,  $F(0)$ , etc.

Row 1:  $F(0)$ ,  $F(0)$ ,  $F(0)$ ,  $F(0)$ ,  $F(0)$ ,  $F(0)$ ,  $F(0)$ , etc.

**Row 2:**  $F(0)$ ,  $F(0)$ ,  $F(0)$ ,  $F(0)$ ,  $F(0)$ ,  $F(0)$ ,  $F(0)$ , etc.

Row 3: F(1), F(0), F(0), F(0), F(1), F(0), F(0), etc.

Combining these together for the rows where there are accesses from both levels give:

F(2), F(0), F(1), F(0), F(2), F(0), F(1), etc.

Obviously for the first scanline and at the edges the number of reads per fragment are much larger and more frequent.

The cache management, address calculation and memory requests are being processed many fragments in advance of the fragments the filter unit is working on (determined largely by the depth of the M FIFO in this unit). So assuming the data is returned back from the memory quick enough it may be possible to have the texel data loaded into the primary cache before it is needed. This can be achieved if the step message collects the texel data as it leaves this unit (in much the same way as occurs in the LB Read Unit and FB Read Unit) but this requires write-through register files (probably not much of an issue) in the Filter Unit but does nothing to help the case where more than one load is needed to fulfil all the new texel data for this step message.

Multiple loads for a step message are common (as outlined above) but typically there are spare load slots on steps which require no new data. We would like to make use of these spare slots otherwise we will take a performance hit on some fragments. For the interior row shown this amounts to 5 cycles for 4 fragments, but the edges will take a bigger hit.

Expedited loading of the cache has been implemented so texel data is loaded in advance of when it is needed, potentially by many cycles. Data returned from the Memory Controller is loaded into the Filter Unit as soon as it is available rather than waiting for the initiating step message.

Information to control the loading of the primary cache is passed to the output stage (called the Dispatcher) in the T FIFO. The step message is passed in a parallel, but independent M FIFO. The Dispatcher will append the new texel data to *any* message, or if no message is going to be sent to the Filter Unit in this cycle it will inject it's own just to load the primary cache.

The expedited loading introduces a few problems of its own which need to be solved to make the scheme viable:

- (1) The expedited texels cannot overwrite texels which may be referenced by step messages which are queued up in the M FIFO until the original texel data has been used. This should be a rare occurrence and only happen when the number of texels used on a scanline is approximately the same as the texture cache can hold.
- (2) Memory latency or just the amount of data required for a step may mean the step

reaches the Dispatcher before all the data has been loaded into the cache so the step message must be delayed.

The solution for (1) adopted is to only update the T FIFO with the expedited load information while there are no steps in the M FIFO (or the current step we are working on which has not been entered into the M FIFO yet) which reference the cache line assigned to be updated.

This entails a FIFO design which can have its valid entries tested for equality to see if any of them use the target cache line. The 72 bits [8 x (8 address bits + 1 valid bit)] of the FIFO width which hold the cache address for each of the 8 texels the step references are available as individual registers and have comparators so the test is done in parallel. The remaining width of the FIFO can be held in a normal FIFO.

Waiting for the offending step(s) to be flushed out of the M FIFO degrades the performance gain we are trying to achieve, and in any case will deadlock when the current step references the cache line we have chosen to replace. Instead we try to find a different cache line which is not referenced by the current step or any queued up in the M FIFO.

Recall the preferred replacement policy is to replace the oldest entry, but in fact we can replace any entry which is not referenced. Which entry should we replace? Some options are:

- We could keep incrementing from the oldest entry looking for the first entry we can replace. This is very simple but suffers from taking several cycles and we are very likely to bump texels one of the following step message would like to use.
- Change the cache policy to be LRU (or something else). Unfortunately this adds significantly to the cost of the cache so isn't really an option.
- Start looking for an unused entry at some offset from the current position, say at half the cache's size from where we are now. If this fails then linearly search until an entry is found (which is always guaranteed as the M FIFO is draining so freeing up cache lines as it goes). This is a good compromise as it doesn't destroy the scanline coherency of the following steps (but may well do so for steps further into the future), should just cost a single cycle in most cases and in the limit is fail safe in that it will wait for the FIFO to drain.

The solution to (2) is for the Dispatcher to maintain a running count of texels loaded into the Filter Unit. As each step message reaches the Dispatcher the running count (called `texelsLoaded` in the behavioral model) is checked against the number of texels

needed to be read by this step. If the texelsLoaded is greater than or equal to what the step needs the step is allowed to proceed to the Filter Unit, otherwise it stalls until sufficient data has been loaded. Once the step is allowed to proceed the texelsLoaded value is decremented by the number of loads the step message was waiting for.

The bottom line is this cache architecture and memory organization is up to 8 times more efficient than the GLINT MX as measured in number of memory reads per output fragment for 1:1 zoom ratio.

### Secondary Cache

The secondary cache, at least compared to the primary cache is a very simple affair. For normal texture mapping it is largely superfluous except in the following cases:

- The texture layout in memory is Linear or Patch64. In these two cases the texture must first be converted to 2x2 patch format before it is loaded into the primary cache. The secondary cache holds the data while this reformatting or aligning is being done. It also allows some re-use of data as the two memory reads needed to build up the 2x2 patch may be able to be used on the next 2x2 patch.
- The texture map is an 8 bit indexed texture map. These are converted into 32 bit textures to be stored in the Filter Unit. The next primary cache load may well use 8 bit texels from the secondary cache rather than having read data from memory.
- The texture data is going to be used for span processing. Span processing does not use the primary cache so the secondary cache is its only way of reducing the memory bandwidth needed.

The secondary cache has four lines where each line holds 128 bits. Why four lines? There are two texture maps and each map can use two memory reads when in Linear or Patch64 layout. The span processing use all four lines to hold up to 512 bits of bit map data, but little re-use would be normally expected - the main gain is reading 128 bits of a font (for example) in one go and extracting several rows worth of bit mask data from this.

The secondary cache is direct mapped (spans use a different algorithm) so the search and replacement policies are very simple and cheap. The cache directory holds addresses (rather than indices as the primary cache does) and these may be logical addresses or physical addresses. An extra bit identifies the type of address so a new logical address cannot alias with an old physical address, for example.

The secondary cache is always enabled and the only use control is to be able to

invalidate it using the InvalidateCache command. This cache should be invalidated whenever texture data has been changed in memory and this data may have been in the secondary cache. (This is never a problem when the Virtual Texture Management changes a texture in memory as the secondary cache holds the logical address and this is invariant unless software re-assigns this logical address to a new texture map. The act of updating the Logical Page Tables through the core will automatically invalidate the secondary cache.)

### **Virtual Texture Management**

Texture maps can be stored in physical memory or in logical/virtual memory. If the texture map is stored in physical memory then it must be physically contiguous and present before that texture is used.

The management of physical textures is complicated by the fact that an application can request more textures than can fit into on-card memory so the textures need to be dynamically swapped, however this is not an easy task to do well because:

- The need to swapping and usage are decoupled in time by the DMA buffers.
- The memory granularity is controlled by the texture map size so is continually changing.
- Memory gets fragmented.
- There is no clear replacement policy.

There are a number of solutions to solving this problem:

- Increase the amount of physical memory to hold texture maps. This is not always possible due to cost or board area constraints and in any case just delays the point at which the problem will re-occur, rather than fixing it altogether.
- Allow textures to be executed out of host memory via the AGP or PCI bus. This is a similar solution to the previous one, except it doesn't have the cost or board area constraints (at least as far as the graphics board is concerned). The downside of this is the bandwidth across the AGP bus is likely to be inferior to the bandwidth out of local memory. Also the latency for the texture data to arrive may degrade texture performance. This method is supported by setting the HostTexture bit in the TextureMapWidth registers. These texture reads will be done across the AGP bus. The PCI bus can be used but because it lacks the efficient random in-page addressing AGP has the texture accesses will be very slow. Note that there may be system reasons why such a method will not work or work poorly. A system with a GLINT

Gamma cannot do this type of access (across AGP) and multiple RX's would require too much bandwidth and not interleave accesses very well.

- The final solution is to treat the texture addresses as logical or virtual addresses. The logical part allows texture maps to be stored in non-contiguous physical pages (a page is 4K bytes). This simplifies the memory management aspect as the granularity now is at the page level. The virtual part allows the dynamic paging of textures out of host or system memory with or without any assistance from the host CPU. This is done on demand so borrows many of the techniques used for CPU memory management. The virtual texture management (of which the logical addressing is a necessary sub-set) is implemented as standard in this unit and will now be described in detail.

Host textures can also be managed; the main difference is that no texture data is downloaded, but is accessed "in situ" using the side band addressing capability of the AGP texture execute mode.

**Mapping an Address**

A brief overview of the sequence of events which occur for a logical texture when the texel causes a primary cache miss will be described. Later on a detailed description will be presented.

- The texel has its logical byte address calculated from it's integer coordinates, base address of the texture, texture map width, etc.
- The logical page the logical address resides in is calculated and the Translation Look aside Buffer (TLB) checked to see if the physical page assigned to the logical page is present. If it is the physical address is formed from the physical page number and the low order bits of the logical address. Note the physical page is relative to the start of the working set and not physical memory. The physical address is then posted to the memory controller.
- If the logical page is not present in the TLB then the Logical Page Table entry for this logical page is read. If the resident bit is set then the logical page is present in the working set and its physical page is read from the Logical Page Table. The TLB is updated so the next time this logical page is accessed the physical page is to hand. The physical address is formed from the physical page number and the low order bits of the logical address and then posted to the memory controller.
- If the logical page is not resident in the working set then details about the page (its host address, target memory pool, etc.) is made available to the host or DMA control-

ler. (The DMA controller is in Gamma for RXs or is integrated into P3.) Sometime later the working set has been updated with the new page of texture data and the Logical Page Table updated to show the faulting logical page is now resident and its physical address. The TLB is updated so the next time this logical page is accessed the physical page is to hand. The physical address is formed from the physical page number and the low order bits of the logical address and then posted to the memory controller.

## Logical Page Mapping

The size of each page is always 4K bytes so the bottom 12 bits of a texel byte address give the byte within a page while the next 16 bits give the page number (the remaining 4 most significant bits are ignored). This gives a maximum virtual texture size of 65536 pages or 256MBytes. The working set can be any number of pages in size. Each logical page has 8 bytes of overhead (in the Logical Page Table) and each physical page has 8 bytes of overhead (in the Physical Page Allocation Table). Some typical sizes for these tables are:

Managed Memory (pages / MBytes)	Table Size
256 / 1MByte	1KBytes
512 / 2MByte	2KBytes
1024 / 4MByte	4KBytes
2048 / 8MByte	8KBytes
4096 / 16MByte	16KBytes
8192 / 32MByte	32KBytes

The Logical Page Table is typically much bigger than the Physical Page Allocation Table. The Logical Page Table must be physically contiguous and is allocated in local buffer memory. The Physical Page Allocation Table must be physically contiguous and is allocated in local buffer memory.







## TLB

The TLB is a fully associative table (or content addressable memory) which caches the recent logical to physical page mappings. It is first check to see if the mapping we want for this page is present as this is much faster than having to query the Logical Page Table in memory. The TLB search happens in a single cycle and is 16 entries for P3 and 64 entries for RX. The replacement policy is oldest first.

A TLB can be classified according to its search policy, its replacement policy and its size. A justification for the chosen attributes will now be given.

The typical search policies are fully associative, set associative and direct mapped. These are graded from most expensive, most flexible (fully associative) to least expensive, least flexible (direct mapped). Set associative and direct mapped both rely on using a subset of address bits to choose one (direct mapped) or a set of locations to search.

The access patterns through a 2D texture map follow an approximate straight line. (It is actually a slightly curved line due to the perspective projection, but this is a minor effect and doesn't change any of the reasoning.) The orientation of the line and its position is arbitrary and successive scanline will all follow on approximately parallel paths. The other variable to contend with is the width of the texture map - this is variable (between texture maps) and a power of two. Given these constraints choosing a set of address bits to which will give a good distribution for each possible orientation of line looks an impossible task. A good distribution is vital otherwise, in the worst case, all addresses along a line could fall into one set (or a single entry for direct mapped) - clearly this will defeat the purpose of a TLB. The fully associative search works equally well in all access patterns.

The common replacement policies are least recently used (LRU), oldest (FIFO), least frequently used and random. The LRU policy usually gives excellent result but is the most expensive, however the approximately regular access patterns repeated from scanline to scanline will make the least recently used page the same as the oldest page (at least within the same polygon). The oldest replacement policy is implemented by a simple counter which selects the entry to replace and is incremented after every replacement. The counter wraps within the available table size.

The size of the TLB is a compromise - the larger the better, but it follows the law of diminishing returns. The minimum useful size is based on the number of pages visited along any path through the texture map. Texture maps are preferably patched 32x32 (a patch at 32 bits per texel is the same size as a page).



Bit No	Name	Description
0-15	Physical Page	These bits hold the physical page number relative to the start of the working set where this logical page is held. If the page is not resident (next field) then these bits are ignored (but will frequently be set to zero). This field is normally maintained by RX, except when the page is marked as a HostTexture.
16	Resident	This bit, when set, marks this logical page as resident in the working set. This field is normally maintained by RX, except when the page is marked as a HostTexture.
17	Host Texture	This bit, when set, marks this logical page as resident in the host memory and it should be accessed using AGP texture execute mode rather than downloading it. The Length field should also be set to zero.
18-31	Reserved	This field is not used but is set to zero whenever the Resident bit is updated.
32-40	Length	This field holds the number of 128 bit words to transfer when a page fault occurs. This allows a page to hold a texture map smaller than 4K without spending the extra download time on the unused words. There is no way to download to unused portion without overwriting the used part. When the physical page is in host memory the length field must be set to zero. This field is maintained by the host.
41-42	Memory Pool	This field holds the memory pool this logical page should be allocated out of. This field is maintained by the host.
43	Virtual Host Page	This bit, when set, indicates the HostPage (next field) is a virtual page in host memory so cannot be accessed directly. Setting this bit will generate an interrupt and involve the host in providing this page of texture data. When this bit is 0 the HostPage is the physical page and will be read directly with no host intervention. This field is maintained by the host.
44-63	Host Page	This field holds the page in host memory where the texture data is held. This is a virtual host page or a physical host page as indicated by the VirtualHostPage bit (previous field). This field is maintained by the host.

The first word in each entry is basically read and written by RX during the memory management activities unless the page is an host texture in which case the host is responsible for the first word as well. The second word is written by the host (either directly via the bypass or via the core using messages) and just read by RX.

The base address of the table is held in the LogicalTexturePageTableAddr register and is aligned to a 64 bit boundary. The number of entries in the table is held in the LogicalTexturePageTableLength register and each logical page number is tested against this limit. If the logical page number is out of range then the address is always mapped into page 0 of the working set and will never cause a texture download. (As a debug aid page 0 of the working set can be missed out of the Physical Page Allocation Table and initialized to some distinctive texture map so any out of range texture mappings cause a distinctive visual effect.) The LogicalTexturePageTableLength is initialized to zero during reset which effectively disabled the logical and virtual texture management.

The table can be updated by the host directly via the bypass once the chip has been synced to make sure there are no conflicting accesses. The Physical Page Allocation Table must also be updated to remove the reference (if any) to the logical page being updated. The TLB should be invalidated incase the updated Logical Page Table has left

any stale data in the TLB. The InvalidateCache command (with bit 2 set) can be used to do this.

The table can also be updated via the normal command stream using the SetLogicalTexturePage command to set the first page to update. The data for bits 32...63 is supplied with the UpdateLogicalTextureInfo command and this will update the Logical Page Table at the previously set page and do all the necessary housekeeping. The logical page to update is auto-incremented so several consecutive table entries are updated. Updates beyond the number of entries in the table (as set by LogicalTexturePageTableLength) are discarded and leave the memory untouched.

The logical table is updated by:

- Memory Allocator to mark a logical page as non resident when its allocated physical page is reclaimed and assigned to another logical address.
- The Download Controller to update the resident bit and physical page field once the download is complete.

**Memory Allocation**

When there is a new page of non host texture data to load into the working set a physical page needs to be allocated to it from the specified pool of memory. The least recently used page in the specified pool is used.

Keeping track of the least recently used page is done by a queue. Whenever a page is first accessed (easily identified by a TLB miss on the page) it is moved to the head of the queue. It therefore follows that the page at the tail of the queue is the least recently used so is the one allocated to the new texture page. This physical page may already be assigned to a logical page so that logical page is marked as non-resident in the Logical Page Table and removed from the TLB. (It is most unlikely it is in the TLB as the working set will normally hold many more pages than the TLB does.)

The queue used to track the physical pages is held in the Physical Page Allocation Table. This table has one entry per physical page and each entry has the following format:

BitNo	Name	Description
0-15	Logical Page	These bits hold the logical page number this physical page has been assigned to. If no assignment has been made (or it has been removed) then the valid bit (next field) will be zero and these bits are ignored (but will frequently be set to zero).
16	Valid	This bit, when set, marks this logical page as resident in the working set. This field is normally maintained by RX.
17-31	Reserved	This field is not used but is set to zero whenever the Resident bit is updated.
32-47	Next Page	This field holds the page number of the next page in the pool - i.e. the next recently used page.
36-63	Previous Page	This field holds the page number of the previous page in the pool - i.e. the previous recently used page.

The Physical Page Allocation Table is not normally accessed by the host. The two exceptions are during power-on initialization and if pages are to be locked down. See later for information on these.

The NextPage and PrevPage fields are used to form a double linked list of the pages assigned to a memory pool. The double linked list is a classic data structure for building queues from as it allows fixed time insertion and deletions. In this application a deletion can occur from any queue entry, but insertions only occur at the head. The head entry is the most recently used physical page and the tail entry is the least recently used page.

A traditional linked list suffers from a linear search time, but by combining it with an array (i.e. table) a constant search time to find a given physical page is guaranteed - you just use the physical page number to index into the table. This is important as a frequent operation is to make a specific physical page the most recent. This involves searching for this page and updating the head (and maybe the tail) pointer to move this page to the head of the queue.

Each memory pool has a head and tail page. These are held in the HeadPhysicalPageAllocation[0...3] and TailPhysicalPageAllocation[0...3] registers respectively and the index relates to each memory pool. These registers are initialized by software at the start of day, but there after are read and written by the hardware.

The PrevPage field for the head page is ignored and will hold links which should be ignored. Similarly for the NextPage field for the tail page.

The maximum size the Physical Page Allocation Table needs to be is the amount of LB memory plus amount of FB memory (in MBytes) divided by 4096. (There is no reason why the Physical Page Allocation Table could not be smaller and just cover the contiguous region set aside for dynamic texture management. Having it cover all the on card memory helps to illustrate some points.) This gives one entry for each 4K page on the card. Many of these pages are not available for virtual texture storage because:

- They hold the color buffers.
- They hold the Z, stencil, etc. buffer.
- They hold the overlay buffers.
- They hold the video overlay buffers.
- They hold non logical textures, icons, fonts, bitmaps, etc.
- They hold the Logical Page Table.
- They hold the Physical Page Allocation Table.
- Run length encoded window ID information.
- They hold logical textures which have been locked down.

These pages are not included in any of the four linked lists so are ignored by the memory allocation hardware.

### **Programming Notes for Non Host Textures**

Following is some general programming information on how the virtual texture management hardware is used.

#### **Start of Day Initialization**

Before any logical or virtual texture management can be done there are a number of areas which need to be initialized (in addition to the usual mode, etc. register initialization):

- Space for the Logical Texture Page Table must be reserved in the local buffer and the table initialized to zero. The LogicalTexturePageAddr and LogicalTexturePageTableLength must be set up.
- Space for the working set must be reserved in the local buffer and/or framebuffer. This need not be physically consecutive pages. The BasePageOfWorkingSet register is set up.

If virtual texture management is to be used then the following additional initialization is required:

- Space for the Physical Page Allocation Table is reserved in the local buffer and PhysicalPageAllocationTableAddr register is set up to point to it.
- Bits 0...31 of each entry in the Physical Page Allocation Table is set to zero - to clear the valid bit.
- Each page entry in the Physical Page Allocation Table is associated to one of the four pools based on which bank of memory it resides in. All the pages in a pool are linked



together as a double linked list by setting the NextPage and PrevPage fields. The order is unimportant, but sequential is simplest. (It will soon get scrambled once the memory allocation has been running for a while.) The PrevPage field for the first entry in the double linked list and the NextPage field for the last entry can be set to any value as they are not used. Finally the HeadPhysicalPageAllocation and TailPhysicalPageAllocation registers for this memory pool are updated with first and last page numbers. Each memory pool is set up like this. (Any number of memory pools up to a maximum of four can be set up. Unused memory pools don't have any pages linked to them and *must* not be referenced in the Logical Texture Page Table.)

The texture management hardware is now ready to be used once logical textures have been created. The texture management can be done on a global basis so all contexts/APIs share the same mechanisms, or can be done on a context by context basis.

### **Creating and Loading Texture Maps**

The sequence of events when the application asks for a texture to be loaded are as follows:

- Host memory to hold the texture map is allocated and locked down. (Virtual host memory could be used, however the driver will need to respond to every page fault and make the textures available in locked physical memory before starting the DMA off to download them. Other than the extra run time overhead and setting the VirtualHostPage flag in the Logical Texture Page Table entries the rest of the operations are the same.) This memory is private to the driver or ICD and not accessible to the application. The pages do not need to be contiguous.
- The logical pages to use for the texture map are allocated from the Logical Texture Page Table. These may be new pages or currently assigned. If they are currently assigned then the texture management hardware will do any necessary housekeeping to prevent aliasing of physical pages to the same logical page (thereby degrading the performance, however still function correctly).
- The host physical page (or host virtual page when host virtual addressing is used) of each page reserved for the texture is found and the HostPage field in for each corresponding entry in the Logical Texture Page Table is updated with it.
- The memory pool this texture is to be stored in is determined and each logical entry has its MemoryPool field set appropriately. (This, in general, is likely to be a difficult thing to determine as the usage of the texture maps is not available. Ideally texture



### **PreLoading Texture Maps**

As mentioned above the texture map is only downloaded when it is used, but it is sometimes useful to ensure it is downloaded when it is created. This can be done by using the *Load* mode to load each logical page in the texture map. Alternatively when a texture map is bound (to a context) you may want to ensure it is resident at this time, rather than wait for a page fault. If the page is already resident then there is no need to load it (as the *Load* mode would do) so the *Touch* mode can be used instead. These can be done using the command *TouchLogicalPages*. This command has the following data fields:

Bit No	Name	Description
0-15	Page	This field set the first Logical Page to touch.
16-29	Count	This field holds the number of pages to touch.
31-31	Mode	This field is set to 3 to touch a page(s) or to 1 to load a page(s).

As each page is touched the corresponding texture data is downloaded.

### **Editing Texture Maps**

To edit the texture map (for example as part of a *TexSubImage* operation in OpenGL) the host's copy is edited. The texture management hardware is notified that the texture pages (if resident) are stale by using the command *TouchLogicalPages* to mark these pages as non resident. This command has the following data fields:

Bit No	Name	Description
0-15	Page	This field set the first Logical Page to mark as stale.
16-29	Count	This field holds the number of pages to mark as stale.
30-31	Mode	This field is set to 0 to mark the pages as stale (i.e. non resident). The primary texture cache is invalidated (using the <i>InvalidateCache</i> command) to ensure it doesn't hold any stale texel data for the texture map just edited.

### **Deleting Texture Maps**

There is no real need to delete texture maps as simply reusing the logical address achieves the same thing. If you really want to delete the pages then the *TouchLogicalPages* command can be used to mark them non resident. (Note that this doesn't mean that these pages are made the least recently used pages so they get reused





Assuming the range of logical pages reserved for host texture management is already included in the length of the Logical Page Table then no further initialization of RX is needed other than to set up the BasePageOfWorkingSetHost register with the address of the region to manage. This is a 256MByte region and can be positioned anywhere in the 4G host address range.

No changes to the Physical Page Allocation Table are needed.

### Creating Logical Texture Maps

The sequence of events when the application asks for a texture to be loaded are as follows:

- Host memory to hold the texture map is allocated and locked down. (Virtual host memory could be used, however the driver will need to respond to every page fault and make the textures available in locked physical memory before starting the DMA off to download them. As these are AGP textures the length field (in the Logical Page Table) is zero so no download actually occurs, however it is convenient to use the same synchronisation methods in the hardware implementation. Other than the extra run time overhead and setting the VirtualHostPage flag in the Logical Texture Page Table entries the rest of the operations are the same.) This memory is private to the driver or ICD and not accessible to the application. The pages do not need to be contiguous.
- The logical pages to use for the texture map are allocated from the Logical Texture Page Table. These may be new pages or currently assigned. If they are currently assigned then the TLB should be invalidated to prevent it from holding stale addresses.
- Each logical page has its physical page, resident and host texture fields in the Logical Page Table updated with the corresponding host physical page where the texture is located. The length field must be set to zero (to disable a download from occurring). The pool field and the hostPage field are not used (but are available to software to hold information about this page).
- The application's texture is copied into the previously allocated host memory and during the copy the texture map is patched and aligned as required by the setting the texture map will be invoked with.

The preferred way to update the Logical Texture Page Table is to use the DownloadAddress and DownloadData commands. The DownloadAddress command takes

the byte address in memory of the Logical Page Table Entry to update. The DownloadData command writes its data to memory and then auto increments the address. Two words are written per logical page entry. After the Logical Page Table has been updated the TLB must be invalidated to prevent it holding stale data (use the InvalidateCache command with bit 2 set) and WaitForCompletion used to ensure the table in memory has been updated before any rendering can start. (The writes to the Logical Page Table are done via the Framebuffer Write Unit so may still be queued up on the subsequent TLB miss, hence stale page data will be read from the Logical Page Table. The WaitForCompletion command ensures this cannot happen.)

Alternatively the Logical Texture Page Table can be edited by software by reading and/or writing it directly to the table in memory by using bypass memory accesses methods. In this case it is the software's responsibility to Sync with the chip first to ensure no outstanding rendering is going to use a logical page about to be updated. The TLB still needs to be invalidated after the bypass updates have been done.

After this set up has been done the texture map can be bound and used.

### **PreLoading Texture Maps**

This is not a meaning full operation with host textures (unless they are virtually managed in which case they can be touched like the non host textures can - see earlier) as the texels are read on demand and not downloaded as pages.

### **Editing Texture Maps**

To edit the texture map (for example as part of a TexSubImage operation in OpenGL) the host's copy is edited. The primary texture cache is invalidated (using the InvalidateCache command) to ensure it doesn't hold any stale texel data for the texture map just edited.

### **Deleting Texture Maps**

There is no real need to delete texture maps as simply reusing the logical address achieves the same thing.

### **Virtual Host Textures**

Virtual host textures are textures which live in virtual host memory so do not need to be locked down into physical memory. As a result they are not guaranteed to be present







bit word has bytes labelled: ABCDEFGH then the three bits swap the bytes as follows:

Bit 2 (long swap)	Bit 1 (short swap)	Bit 0 (byte swap)	swapped ABCDEFGH
0	0	0	ABCDEFGH
0	0	1	BADCFEHG
0	1	0	CDABGHEF
0	1	1	ABDCEFGH
1	0	0	EFGHABCD
1	0	1	FEHGBACD
1	1	0	GHEFCDAB
1	1	1	HGFEDCBA

- Next the texel is optionally mirrored. This is controlled by the TextureReadMode0.Mirror bit. The mirror swaps bits:  
(0, 63), (1, 62), (2, 61),... (31, 32).
- The texel is next optionally inverted under control of the TextureReadMode0.Invert bit.
- When TextureReadMode0.OpaqueSpan is zero the texel is ANDed with the pixel mask to remove pixels from the mask. When TextureReadMode0.OpaqueSpan is one the texel is ANDed with the color mask (in the SpanColorMask message) to control foreground/background color selection.

Windows normally supplies its bitmasks as a byte stream with successive bytes controlling 8 pixel groups at increasing x (i.e. towards the right edge). Bit 7 within a byte controls the left most pixel (for that group) and bit 0 the right most pixel. To match up the pixel mask order (bit 0 controls the left most pixel, bit 63 the right most pixel) the three byte swap bits are all set and the mirror bit set.

### Indexed Textures

Indexed textures are a special case because they are stored as 8 bit texels and expanded to 32 bit texels when loaded into the Texture Filter Unit (the expansion happened in the Texture LUT Unit). This makes the addressing and cache management slightly more complicated as the addressing uses the 8 bit texel size, while the cache management uses the 32 bit texel size.

The secondary cache holds the texture data in its 8 bit format so reduces the number of memory reads when the access path is mainly in u across the texture map.





Following are some details of a sample implementation.

**Overview**

A block diagram of the unit is shown in **Figure 10**. The overall unit is split into 7 sub-units and these are basically organized into three groups:

The Primary Cache Manager, Address Generator and Dispatcher form the core of the unit and work in a similar way to the other read units. The logical address translation is handled by the Address Mapper and TLB. The dynamic texture loading is handled by the Memory Allocator and the Download Controller.

The interfaces between all the units are shown as FIFOs, but most of the FIFOs are just a register with full/empty flags for simple handshaking. The single deep FIFOs have been used as they clearly delineate the functionality between units and allow a single sub unit to be responsible for a single resource.

The two shared resources which are managed in this way are the TLB and Memory Allocator. The TLB is mainly queried by the Address Mapper but the Memory Allocator needs to invalidate pages when a physical page is re-assigned. The Memory Allocator will allocate pages when requested by the Download Controller, but also needs to mark pages as "most recently used" when requested by the Address Mapper.

There are two read/write ports to the Memory Controller used to access the Logical Page Table and the Physical Page Allocation Table - these are 64 bit ports and are not FIFO buffered. There is no point in trying to queue up reads or writes on these ports as the texture process stalls until these operations are satisfied.

The read port to the Memory Controller is used to read texture data and has a deep address FIFO and return data FIFO to absorb latency.

The write port to the Memory Controller is used by the Download Controller to write texture data into memory during a download. The path from the Texture Input FIFO to the Memory Controller is 128 bits wide so the maximum download bandwidth can be sustained.

All the controlling registers (TextureReadMode, TextureMapWidth, TextureBaseAddr, etc. are all held in the Primary Cache Manager so the responsibility for loading them from the message stream, context dumping and readback is all concentrated in one place. This does mean that before any of them can be updated any outstanding work which may depend on them has to be allowed to complete. To make things simpler before any of these registers (see behavioral model for a full list) is updated the all the sub units need to









- Memory Allocator Port
- Address Mapper Port
- Texture Write Port
- Texture Read Port

Note that the first two ports are not FIFO buffered, so they will block subsequent texture processing until their read or write request have been serviced.

### **Texture Read Port**

This port is used to read texel data from memory. The addresses (after any necessary translation) are written into the Tx Addr FIFO and sometime later the 128 bits worth of data are returned via the Tx Data FIFO.

The following information is passed to the Memory Controller in a FIFO:

Bit No.	Name	Width	Description
0-1	Type	2	Indicates what the target memory is. The options are: 0 = FB Memory 1 = LB Memory 2 = PCI
2-29	Addr	28	The read address of the 128 bits of memory data.

The following information is passed back from the Memory Controller in a FIFO:

Bit No.	Name	Width	Description
0-127	Data	128	The data read from the memory.

### **Texture Write Port**

This port is used by the Download Controller to write texture data into its allocated physical page. It is also used to update the Logical Texture Page Table to mark the page as being resident once it has been downloaded.

The following information is passed to the Memory Controller in a FIFO:

Bit No.	Name	Width	Description
0-1	Type	2	Indicates what the target memory is. The options are: 0 = FB Memory 1 = LB Memory 2 = PCI
2-29	Addr	28	The write address of the 128 bits of memory data.
30-45	ByteEnables	16	A high on a bit enables that byte to be written. The 1st byte enable corresponds to data bits 0-7.
46-173	Data	128	The data to be written to the memory.

The following information is passed back from the Memory Controller:

Bit No.	Name	Width	Description
0	TrWrComplete	1	This signal is asserted by the memory controller when the FIFO is empty and <i>all</i> writes from this port, the Memory Allocator Port and the Address Mapper Port have been written to memory so can be read from another port.

### **Memory Allocator Port**

This port is used to update the Logical Texture Page Table with information from the host and to remove references from a physical page to a logical page in the Physical Page Allocation Table. The port is 64 bits wide (to save routing a 128 bit data bus from the Memory Controller). The read and write operations are buffered by a single level FIFO (to provide a simple interface) so will stall until their operations are satisfied.

The following signals are passed to the Memory Controller (MC):

Bit No.	Name	Width	Description
0-1	Type	2	Indicates what the target memory is. The options are: 0 = FB Memory 1 = LB Memory 2 = PCI
2	Command	1	0 = Write, 1 = Read
3-31	Addr	29	The write address of the 64 bits of memory data.
32-39	ByteEnables	8	A high on a bit enables that byte to be written. The 1st byte enable corresponds to data bits 0-7.
40-103	WrData	64	The data to be written to the memory.

The following signals are passed from the Memory Controller (MC):

Bit No.	Name	Width	Description
0	RdData	64	The data read from memory



132-143	f0j1	This field holds j1 index for texture 0, even mip maps or even slices for 3D textures. The least significant bit of the computed index is not needed so the original 12 bit number has been reduced to 11 bits.
144-147	T0Valid T1Valid T2Valid T3Valid	These bits show which texels are valid texels as a function of the filter type and the map type (1D or 2D) and will limit the addresses checked in the primary cache and hence any texture reads ultimately done.
148-151	T0BorderColor T1BorderColor T2BorderColor T3BorderColor	These bits show which texels are to use the border color instead of texel data. These are only taken into account for valid combinations of indices (see previous field).
152-155	f0map	This field holds the map level the texels (T0...T3) are on.
156-167	f1i0	This field holds i0 index for texture 1, odd mip maps or odd slices for 3D textures. The least significant bit of the computed index is not needed so the original 12 bit number has been reduced to 11 bits.
168-179	f1i1	This field holds i1 index for texture 1, odd mip maps or odd slices for 3D textures. The least significant bit of the computed index is not needed so the original 12 bit number has been reduced to 11 bits.
180-191	f1j0	This field holds j0 index for texture 1, odd mip maps or odd slices for 3D textures. The least significant bit of the computed index is not needed so the original 12 bit number has been reduced to 11 bits.
192-203	f1j1	This field holds j1 index for texture 1, odd mip maps or odd slices for 3D textures. The least significant bit of the computed index is not needed so the original 12 bit number has been reduced to 11 bits.
204-207	T4Valid T5Valid T6Valid T7Valid	These bits show which texels are valid texels as a function of the filter type and the map type (1D or 2D) and will limit the addresses checked in the primary cache and hence any texture reads ultimately done.
208-211	T0BorderColor T1BorderColor T2BorderColor T3BorderColor	These bits show which texels are to use the border color instead of texel data. These are only taken into account for valid combinations of indices (see previous field).
212-215	f1map	This field holds the map level (T4-T7) are on.
216-224	i0	Interpolation coefficient between (T0, T1) and (T2, T3) in 1.8 unsigned fixed point format.
225-233	i1	Interpolation coefficient between (T0, T2) and (T1, T3) in 1.8 unsigned fixed point format.
234-242	i2	Interpolation coefficient between (T4, T5) and (T6, T7) in 1.8 unsigned fixed point format.
243-251	i3	Interpolation coefficient between (T4, T6) and (T5, T7) in 1.8 unsigned fixed point format.
252-260	i4	Interpolation coefficient between (T0, T1, T2, T3) and (T4, T5, T7, T7) in 1.8 unsigned fixed point format.















TextureDownloadInProgress	1	This is asserted by the Download Controller and is used to validate the DownloadLogicalPage value. The Address Mapper uses this to check if the download it wants is currently being done.
DownloadLogicalPage	16	This is set by the Download Controller to identify the logical page it is in the process of downloading.
TextureDownloadComplete	1	This is asserted by the Download Controller when it has finished downloading the texture the Address Mapper is waiting on.

### **Dispatcher**

The Dispatcher holds the data part of the secondary cache and forwards texel data to the primary cache (in the Filter Unit). Texel data is allowed to flow through whenever it arrives from the Memory Controller, but under control from commands received via the T FIFO. A count of the texel data loaded for each filter bank (i.e. texture map) is maintained so that an active step message can be delayed until all the texel data it requires is present in the Filter Unit. In normal operation this delay should not be invoked very often.

The Dispatcher also handles span processing. This involves zero extending the texel data to a 64 bit bitmask, byte swapping, mirroring and inverting when necessary and finally anding the pixel mask in the span step message.

### **Texture DMA Controller**

When a texture page fault occurs the Texture Read Unit interfaces with a Texture DMA Controller to actually get the data. This DMA Controller is in Gamma for a RX based system, or in P3 for a P3 system.

The P3 Texture DMA Controller just handles a single request at a time. The Gamma based Texture DMA Controller is monitoring multiple RXs and broadcasts the texture data to *all* RXs and not just the requesting one.

The following hardware signals are used to communicate between the Texture Read Unit and the Texture DMA Controller (each RX will provide its own pair of signals and a mechanism to allow the texture data to be broadcast to all RXs simultaneously):

- pciTextureDownloadRequest. This signal is asserted by Texture Read Unit to request a texture download. It is de-asserted once the texture download has started.
- TextureFIFOFull. This signal is asserted by the Texture Read Unit when it is not able to accept any more data being written into the TextureInput FIFO.

When the Texture DMA Controller has detected a download request it reads three PCI registers from the requester. These registers are:

- HostTexturePage. This register holds the host page (in bits 0...19) where the texture

resides. This is either a physical page or a virtual page. A bit in the TextureOperation register identifies the type of page. If the page is a virtual page then an interrupt is generated and the host will read the page and initiate the DMA once the data has been made available. The conversion from page to address is done by multiplying by 4096.

- **LogicalTexturePage.** This register holds the logical page for the texture data and is returned back to the Texture Read Unit in bits 0...15 of the first entry written to the Texture Input FIFO (the FIFO is 128 bits wide) as a header preceding the actual texture data. (All 32 bits of the register are returned in bits 0...31 to allow for future capabilities.) In a multi-RX system all the RXs take the texture download and not just the RX which requested it.
- **TextureOperation.** This register holds the following information:

Bit No.	Name	Description
0-8	Length	Transfer length in multiples of 128 bit words, maximum being 256
9-10	Memory Pool	Identifies which memory pool the physical page is to be allocated from.
11	HostVirtual Address	This bit, when set, indicates the address is a host virtual address so the data cannot be read directly without software intervention. The TextureDownload interrupt is generated, if enabled.

This data (and bits 12-31) are returned back to the Texture Read Unit in bits 32-64 of the first entry written to the Texture Input FIFO (the FIFO is 128 bits wide) as a header preceding the actual texture data.

Gamma broadcasts the LogialTextureAddress and TextureOperation words to the TextureInput FIFO before the actual texture data. The Texture Read Unit on seeing this information will remove any TextureDownloadRequest this transfer will satisfy and allocate space in its texture working set for the new texture page.

The three PCI registers need to be offset from their base address based on the RX number.

If the texture download request results in a TextureDownload interrupt being generated the TextureAddr PCI register is loaded with the virtual address and the TextureOperation PCI register is loaded with the TextureOperation data read from Texture Read Unit before the interrupt is generated. The host services the interrupt, reads these two registers and provides the data. When the data is available in memory the physical address where the data is located is written in to the TextureAddr PCI register. This will wake up the texture download DMA controller and it will do the download.



### Texture DMA Controller

```
void TextureDMAController (void)
{
    // These three registers can also be read and written by the host across
    // the PCI bus.
    uint32    regHostTextureAddr, regLogicalTexturePage, regTextureOperation;

    uint128    fifoData;
    uint9      length;

    forever
    {
        if (pciTextureDownloadRequest is asserted)
        {
            // Get the texture request info from the Texture Read Unit.
            regHostTextureAddr = pciHostTexturePage << 12;
            regLogicalTexturePage = pciLogicalTexturePage;
            regTextureOperation = pciTextureOperation;

            if (textureOperation.VirtualHostAddress)
            {
                // Host virtual address. Just raise an interrupt and wait for
                // the host to kick off the DMA.
                SetInterrupt (eTextureDownload);

                // Host responds when it is ready by writing to the
                // regHostTextureAddr when it is ready.
                while (no write to regHostTextureAddr)
                    ; // wait

                // Now regHostTextureAddr holds the physical addr supplied by
                // host;
            }

            // SlaveTextureDownload is a bit in a general PCI register.
            if (SlaveTextureDownload == 0)
            {
                bits 0...31 of fifoData = regLogicalTexturePage;
                bits 32...63 of fifoData = regTextureOperation;
                bits 64...127 of fifoData = 0;
                WriteTextureFIFO (fifoData);

                // Wait for the texture request to be removed before sending
                // texture data.

                while (pciTextureDownloadRequest is asserted)
                    ; // wait.

                // Transfer the data.
                length = bits 0...8 of regTextureOperation;
                while (length > 0 && pciCommandMode.TextureDownloadEnalbe)
                {
                    bits 0...31 of fifoData = ReadAddr (regHostTextureAddr + 0);
```

```

        bits 32...63 of fifoData = ReadAddr (regHostTextureAddr + 4);
        bits 64...95 of fifoData = ReadAddr (regHostTextureAddr + 8);
        bits 96...127 of fifoData = ReadAddr (regHostTextureAddr + 12);
        WriteTextureFIFO (fifoData);
        length--;
        regHostTextureAddr += 16;           // byte address
    }
}
}
}
}

```

```

void WriteTextureFIFO (int128 data)
{
    Wait for room in the Texture Input FIFO;
    Write data into Texture Input FIFO;
}

```

```

uint32 ReadAddr (uint32 byteAddr)
{
    return 32 bits of data read from byteAddr;
}

```

### **RX Texture DMA Controller**

```

void TextureDMAController (void)
{
    // These three registers can also be read and written by the host across
    // the PCI bus.
    uint32    regHostTextureAddr, regLogicalTexturePage, regTextureOperation;

    uint32    data;
    uint9     length;
    int3      i = 0;
    int       kRXCount;    // Holds the number of RX in the system

    forever
    {
        if (pciTextureDownloadRequest[i] is asserted)
        {
            // Get the texture request info from the Texture Read Unit.
            regHostTextureAddr = ReadTextureInfo (i, 0) << 12;
            regLogicalTexturePage = ReadTextureInfo (i, 1);
            regTextureOperation = ReadTextureInfo (i, 2);

            if (textureOperation.VirtualHostAddress)
            {
                // Host virtual address. Just raise an interrupt and wait for
                // the host to kick off the DMA.
                SetInterrupt (eTextureDownload);

                // Host responds when it is ready by writing to the
                // regHostTextureAddr when it is ready.
                while (no write to regHostTextureAddr)
                {
                    // wait
                }
            }
        }
    }
}

```

```

        // Now regHostTextureAddr holds the physical addr supplied by
        // host;
    }

    bits 0...31 of fifoData = regLogicalTexturePage;
    bits 32...63 of fifoData = regTextureOperation;
    bits 64...127 of fifoData = 0;
    WriteTextureFIFO (fifoData);

    // Wait for the texture request to be removed before sending
    // texture data.

    while (pciTextureDownloadRequest[i] is asserted)
        ; // wait.

    // Transfer the data.
    length = bits 0...8 of regTextureOperation;
    while (length > 0 && pciCommandMode.TextureDownloadEnalbe)
    {
        fifoData = ReadAddr (regHostTextureAddr + 0);
        WriteTextureFIFO (aata);
        fifoData = ReadAddr (regHostTextureAddr + 4);
        WriteTextureFIFO (aata);
        fifoData = ReadAddr (regHostTextureAddr + 8);
        WriteTextureFIFO (aata);
        fifoData = ReadAddr (regHostTextureAddr + 12);
        WriteTextureFIFO (aata);

        length--;
        regHostTextureAddr += 16; // byte address
    }

    // Round robbin to the next RX.
    i++;
    if (i == kRXCount)
        i = 0;
}

uint32 ReadAddr (uint32 byteAddr)
{
    return 32 bits of data read from byteAddr;
}

// Reading the TextureFIFO returns the info (saves on address decode and
// registers. Note this register is overloaded onto the XXX register.

int32 ReadRXTextureInfo (int3 rxID, int2 register)
{
    int32 addr, data;
    addr = pciRXTextureBase + rxID * 12 + register * 4; // byte addr.

```



```

    data = PCI read on the secondary pci bus to addr;
    return data;
}

void WriteTextureFIFO (int32 data)
{
    int32 i;
    int32 addr;

    for (i = 0; i < kRXCount; i++)
    {
        while (TextureInputFIFOFull[i] is asserted)
            ; // wait until it goes empty.
    }

    // Increment the address to allow PCI bust writes.
    addr = pciRXTextureFIFOBase + textureDownloadOffset * 4;
    Write data to addr on the secondary PCI bus;

    textureDownloadOffset++; // wraps for modulo indexing
}

```

## General Control

This unit is controlled by the TextureReadMode0 and TextureReadMode1 messages. These have identical fields (although some fields are ignored in TextureReadMode1). Not all combinations of modes across both registers are supported and where there is a clash the modes in TextureReadMode0 take priority. For per pixel mip mapping the TextureRead0 and TextureReadMode1 register should be set up the same as should the TextureMapWidth0 and TextureMapWidth1 registers.

BitNo	Name	Description
0	Enable	When set causes any texels needed by the fragment, but not in the primary cache to be read. This is also qualified by the TextureEnable bit in the PrepareToRender message.
1-4	Width	This field holds the width of the map as a power of two. The legal range of values for this field is 0 (map width = 1) to 11 (map width = 2048). This is only used when Texture3D is enabled and then is only used for cache management purposes and <i>not</i> for address calculations. Note this field is ignored in TextureReadMode1.
5-8	Height	This field holds the height of the map as a power of two. The legal range of values for this field is 0 (map height = 1) to 11 (map height = 2048). This is only used when Texture3D is enabled and then is only used for cache management purposes and <i>not</i> for address calculations. Note field bit is ignored in TextureReadMode1.
9-10	TexelSize	This field holds the size of the texels in the texture map. The options are: 0 = 8 bits 1 = 16 bits 2 = 32 bits 3 = 64 bits (Only valid for spans)
11	Texture3D	This bit, when set, enables 3D texture index generation. Note this bit is ignored in TextureReadMode1. The CombinedCache mode bit should not be set when 3D textures are being used.

12	Combine Cache	This bit, when set, causes the two banks of the Primary Cache to be joined together, thereby increasing the size of a single texture map which can be efficiently handled. Note this bit is ignored in TextureReadModel
13-16	MapBase Level	This field defines which TextureBaseAddr register should be used to hold the address for map level 0 when mip mapping or the texture map when not mip mapping. Successive map levels are at increasing TextureBaseAddr registers upto (and including) the MaxMaxLevel (next field). 3D textures always use TextureBaseAddr0.
17-20	MapMax Level	This field defines the maximum TextureBaseAddr register this texture should use when mip mapping. Any attempt to use beyond this level will clamp to this level.
21	LogicalTexture	This bit, when set, defines this texture or all mip map levels, if mip mapping, to be logically mapped so undergo logical to physical translation of the texture addresses.
22	Origin	This field selects where the origin is for a texture map with a Linear or Patch64 layout. The options are: 0 = Top Left. 1 = Bottom Left A Patch32 or Patch2 texture map is always bottom left origin.
23-24	Texture Type	This field defines any special processing needed on the texel data before it can be used. The options are: 0 = Normal. 1 = Eight bit indexed texture. 2 = Sixteen bit YVYU texture in 422 format. 3 = Sixteen bit VYUY texture in 422 format.
25-27	ByteSwap	This field defines the byte swapping, if any, to be done on texel data when it is used as a bitmap. This is automatically done when spans are used. Bit 27, when set, causes adjacent bytes to be swapped, bit 26 adjacent 16 bit words to be swapped and bit 27 adjacent 32 bit words to be swapped. In combination this byte swap the input (ABCDEFGH) as follows: 0 ABCDEFGH 1 BADCFEHG 2 CDABGHEF 3 ABCDEFGH 4 EFGHABCD 5 FEHGBADC 6 GHEFCDAB 7 HGFEDCBA
28	Mirror	This bit, when set will mirror any bitmap data. This only works for spans.
29	Invert	This bit, when set will invert any bitmap data. This only works for spans.
30	Opaque Span	This bit, when set, will cause the SpanColorMask to be modified rather than the pixel mask in SpanStepX or SpanStepYDom messages.

The `TextureCacheReplacementMode` register controls the replacement policy in the primary cache. It has the following fields:

Bit No	Name	Description
0	Keep Oldest0	This bit, when set, will keep the oldest texels on the scanline when the cache bank 0 is about to wrap and just re-use a set of scratch lines.
1-5	Scratch Lines0	This field holds the number of cache lines to use as scratch lines when the cache bank 0 wraps and the KeepOldest mode bit is set. The value in this field has a MIN_SCRATCH_SIZE value (currently 8) added to it so we can guarantee the scratch line size can always accommodate the cache lines the current fragments requires with some left over. Failure to make this provision would lead to deadlock.
6	Keep Oldest1	This bit, when set, will keep the oldest texels on the scanline when the cache bank 1 is about to wrap and just re-use a set of scratch lines.







## CLAIMS

What is claimed is:

1. A computer system, comprising:  
a graphics accelerator unit which manages page faulting of texture data invisibly to the host processor.
2. A computer system, comprising:  
a graphics accelerator unit which manages page faulting of texture data, from dedicated graphics memory into a main memory used by at least one host processor, invisibly to the host processor, except when said graphics accelerator unit calls  
5 for data which has not recently been present in said main memory.
3. A computer system, comprising:  
at least one CPU, operatively connected to have read/write access to a main memory;  
first memory management logic, which virtualizes said main memory with reference to  
at least one bulk storage unit; and  
5 a graphics accelerator unit, comprising rendering accelerator logic, dedicated graphics memory, and a second memory management unit which  
manages texture data for said accelerator logic and  
performs page faulting of said texture data,  
invisibly to said CPU.

[illegible][illegible]

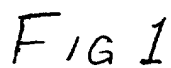


FIG 1





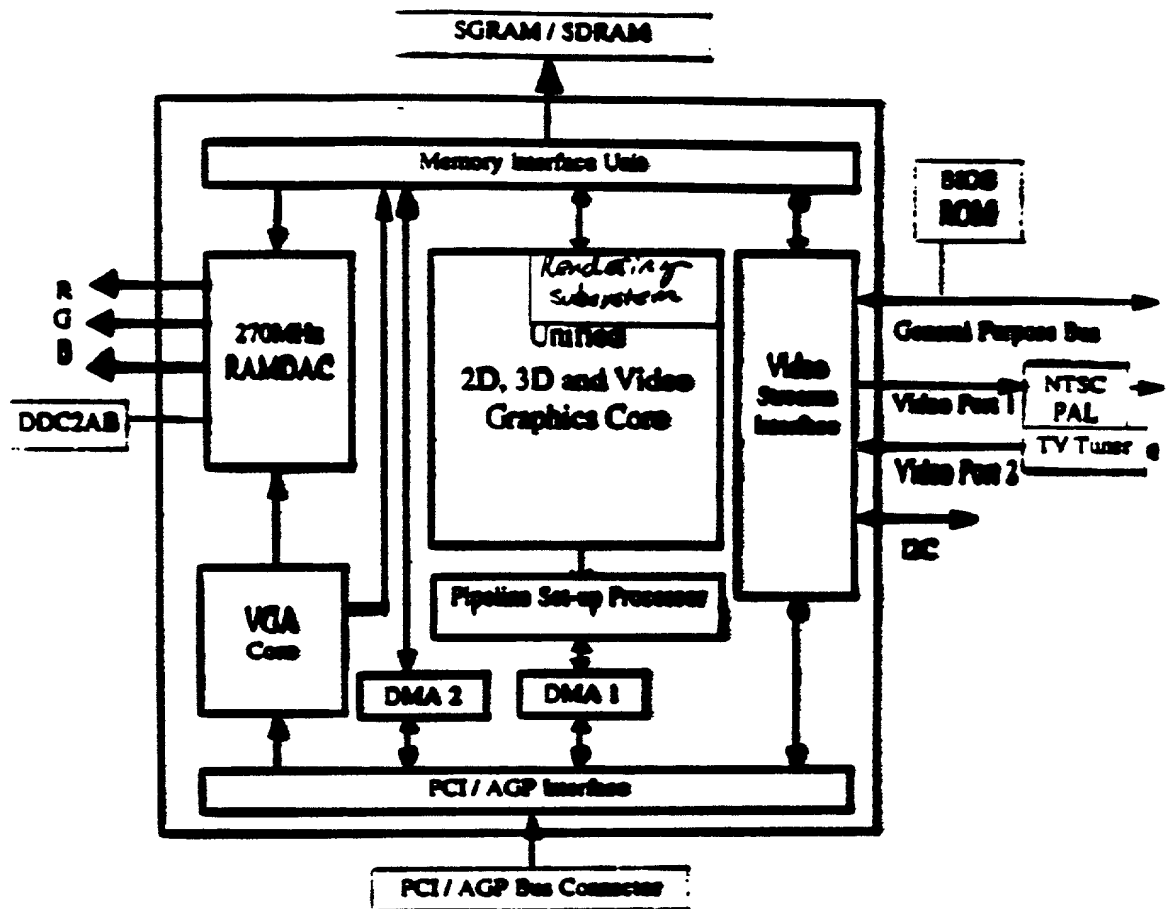


Figure 3



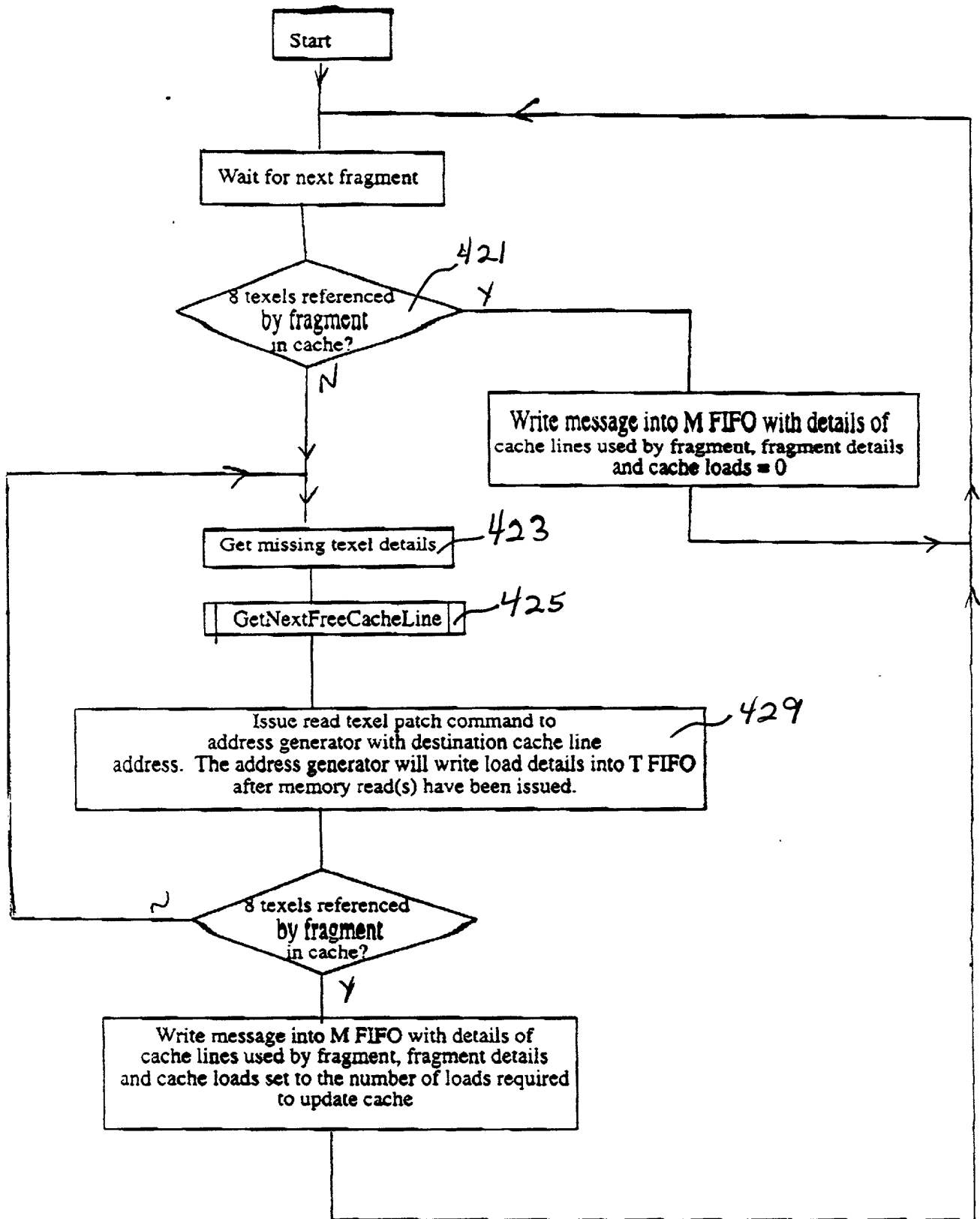


FIG 4B

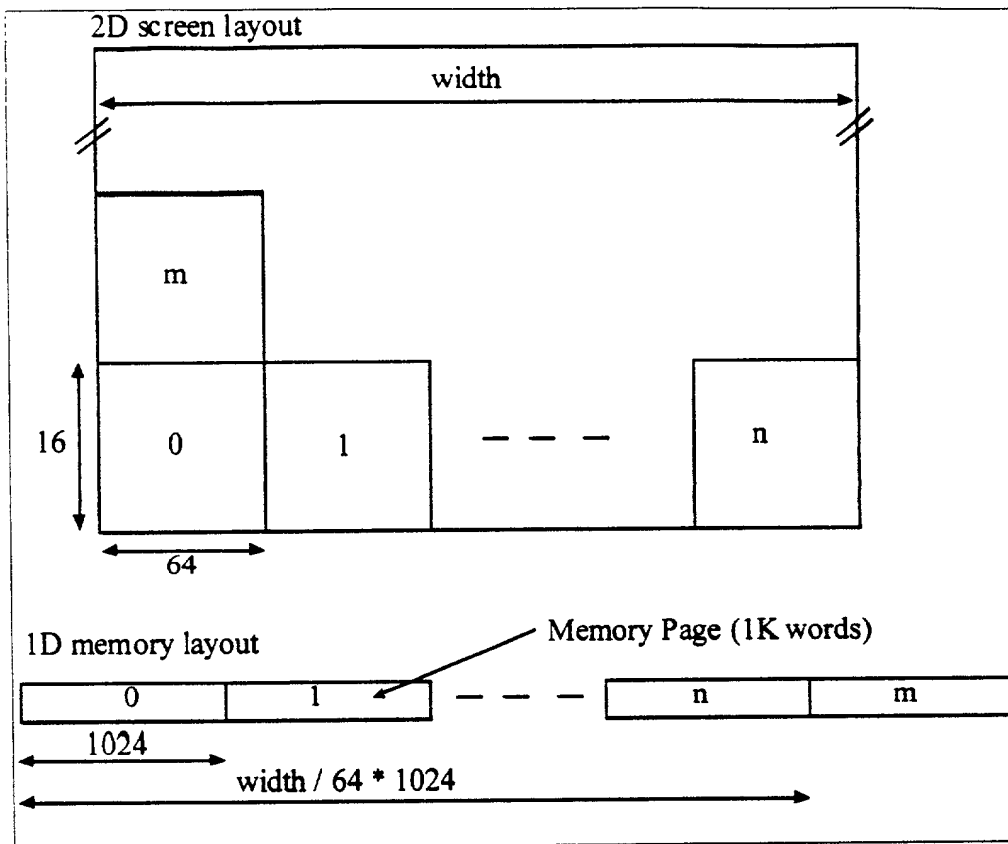


FIG. 5

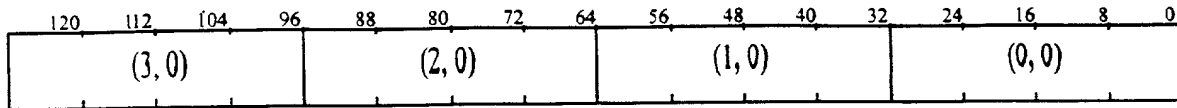
T0 (0,4)	T1 (1,4)	T0 (2,4)	T1 (3,4)	T0 (4,4)	T1 (5,4)	T0 (6,4)	T1 (7,4)	T0 (8,4)	T1 (9,4)
T2 (0,3)	T3 (1,3)	T2 (2,3)	T3 (3,3)	T2 (4,3)	T3 (5,3)	T2 (6,3)	T3 (7,3)	T2 (8,3)	T3 (9,3)
T0 (0,2)	T1 (1,2)	T0 (2,2)	T1 (3,2)	T0 (4,2)	T1 (5,2)	T0 (6,2)	T1 (7,2)	T0 (8,2)	T1 (9,2)
<del>T0 (0,1)</del>	<del>T1 (1,1)</del>	<del>T2 (2,1)</del>	<del>T3 (3,1)</del>	<del>T2 (4,1)</del>	<del>T3 (5,1)</del>	<del>T2 (6,1)</del>	<del>T3 (7,1)</del>	<del>T2 (8,1)</del>	<del>T3 (9,1)</del>
<del>T0 (0,0)</del>	<del>T1 (1,0)</del>	<del>T2 (2,0)</del>	<del>T3 (3,0)</del>	<del>T0 (4,0)</del>	<del>T1 (5,0)</del>	<del>T0 (6,0)</del>	<del>T1 (7,0)</del>	<del>T0 (8,0)</del>	<del>T1 (9,0)</del>

- 32 bit texels in memory word
- 16 bit texels in memory word
- 8 bit texels in memory word

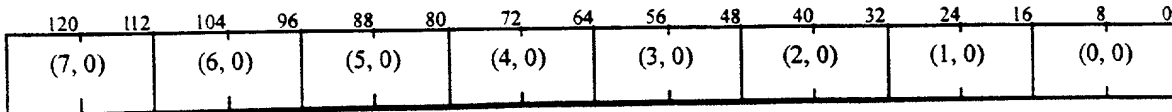
FIG. 6

## Linear or Patch64 Memory Layouts

32 bits per texel



16 bits per texel



8 bits per texel

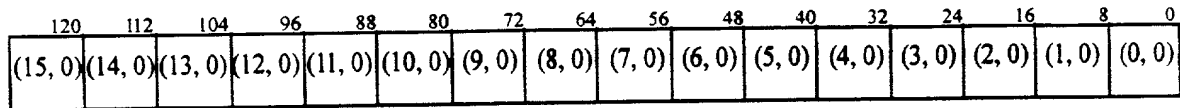
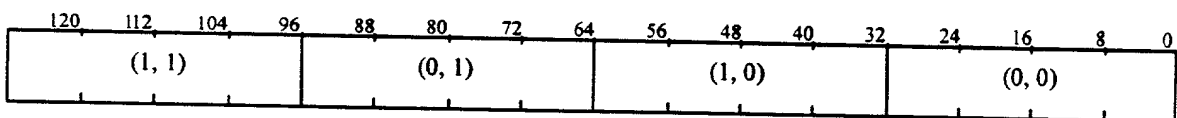


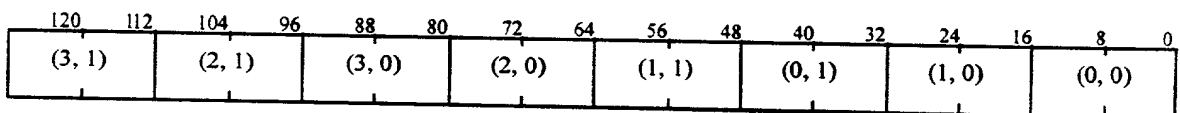
FIG. 7A

## Patch32\_2 or Patch2 Memory Layouts

32 bits per texel



16 bits per texel



8 bits per texel

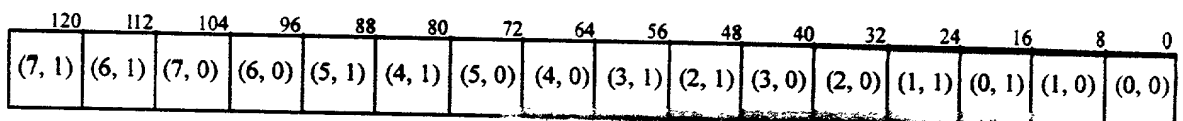


FIG. 7B





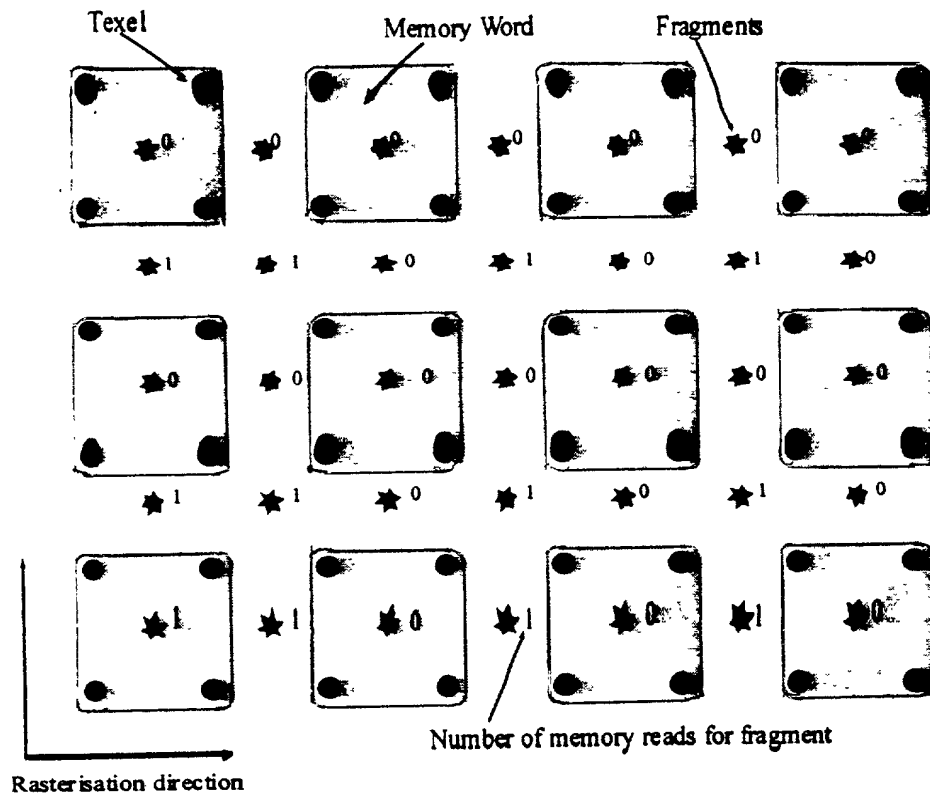


FIG. 9

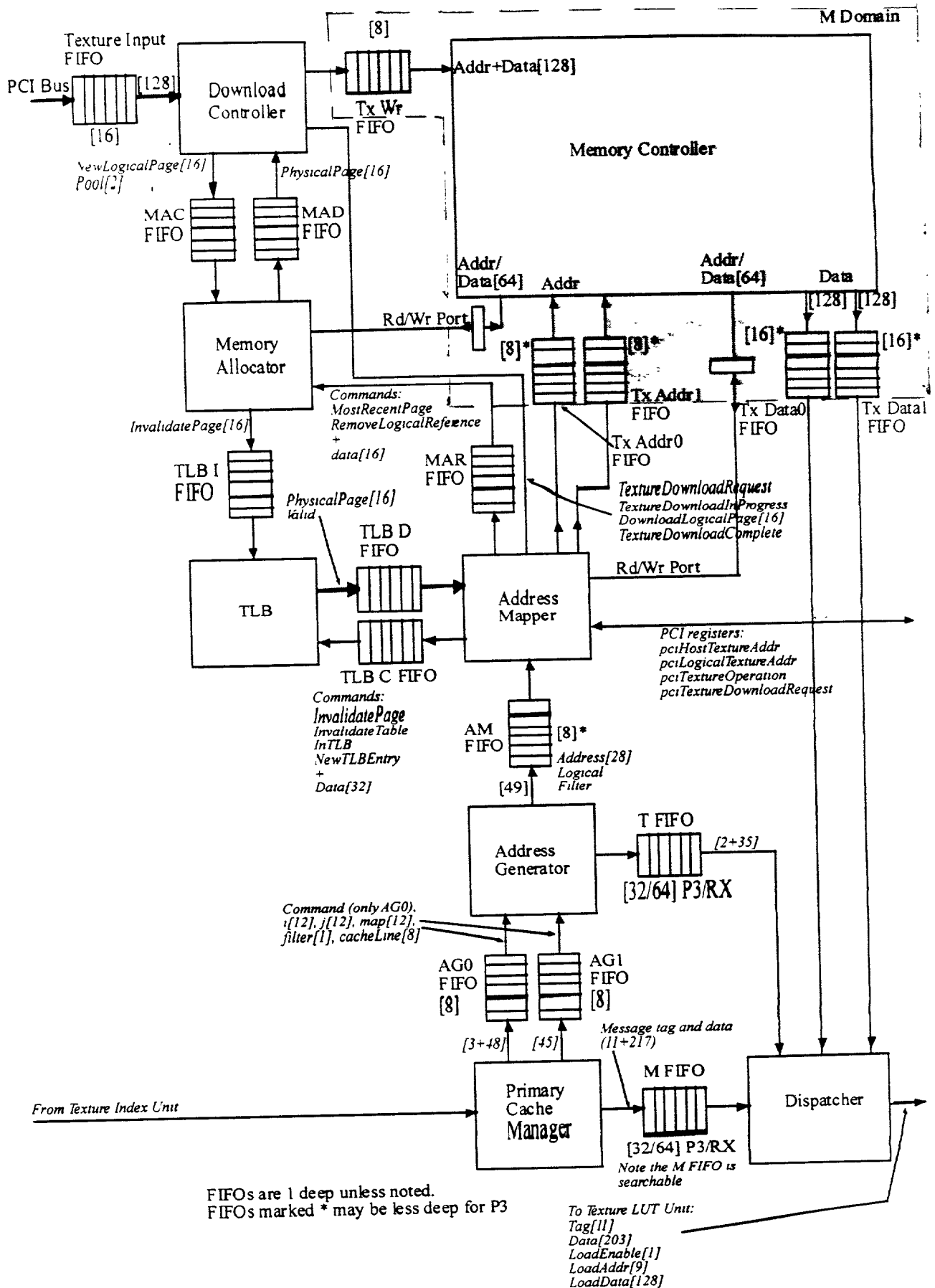


FIG. 10

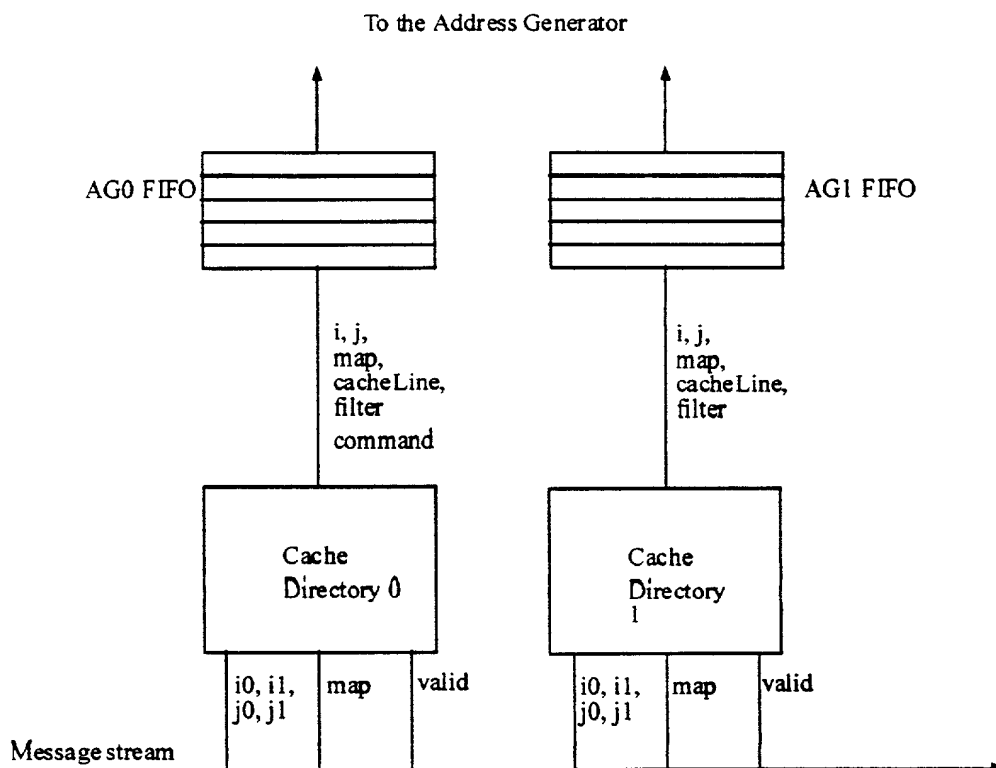


FIG. 11

The diagram illustrates a CAM (Content-Addressable Memory) architecture. It features a central CAM array with rows indexed 0 to 63. Each row contains a CAM Cell. The input keys (k0, k1, k2, k3) are connected to the CAM cells via a multiplexer (mux). The output of the CAM cells is connected to a LoadData bus. The CAM cells are also connected to a LoadEnable bus and an Invalidate bus. The CAM cells are connected to a cache (cacheAddr0 to cacheAddr3) via an inCache bus. The cache is connected to a 6 (P3) 8 (RX) bus.

FIG 12

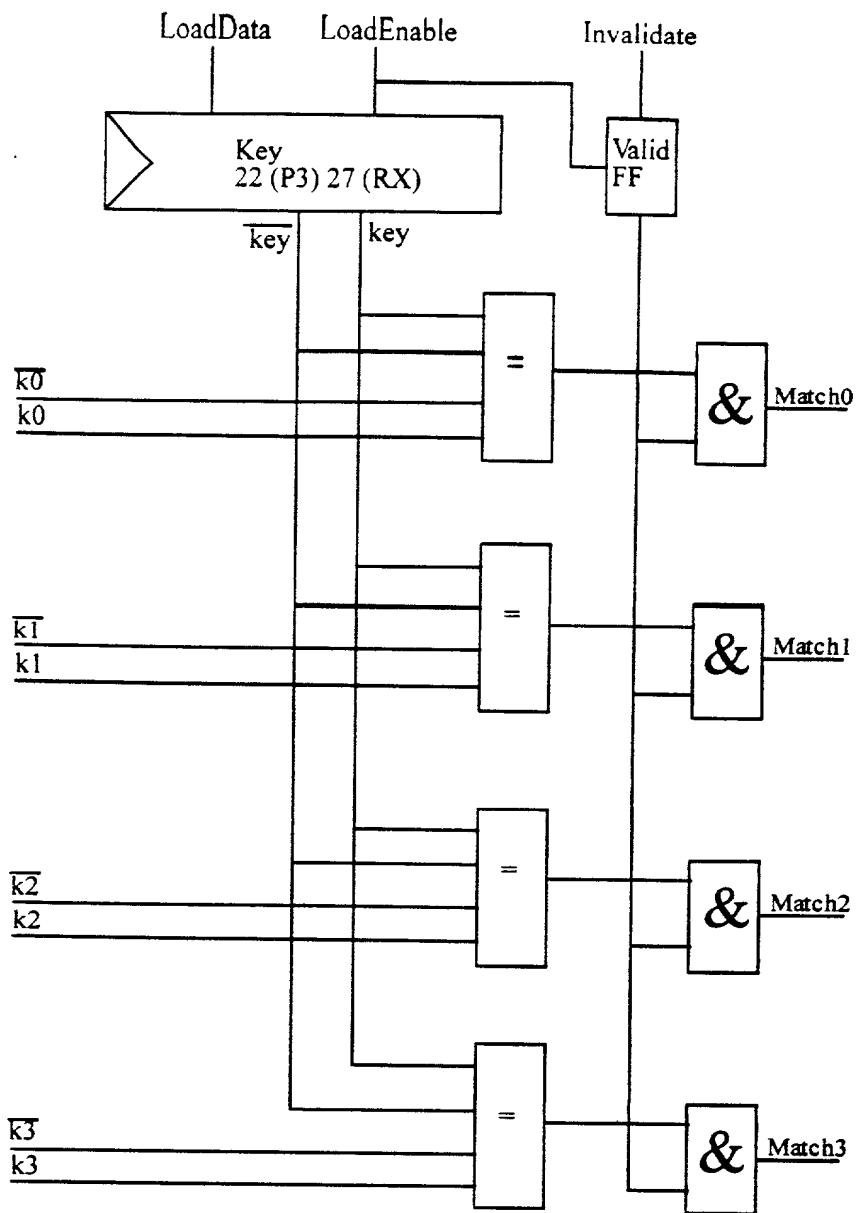


FIG. 13

000050 5224550

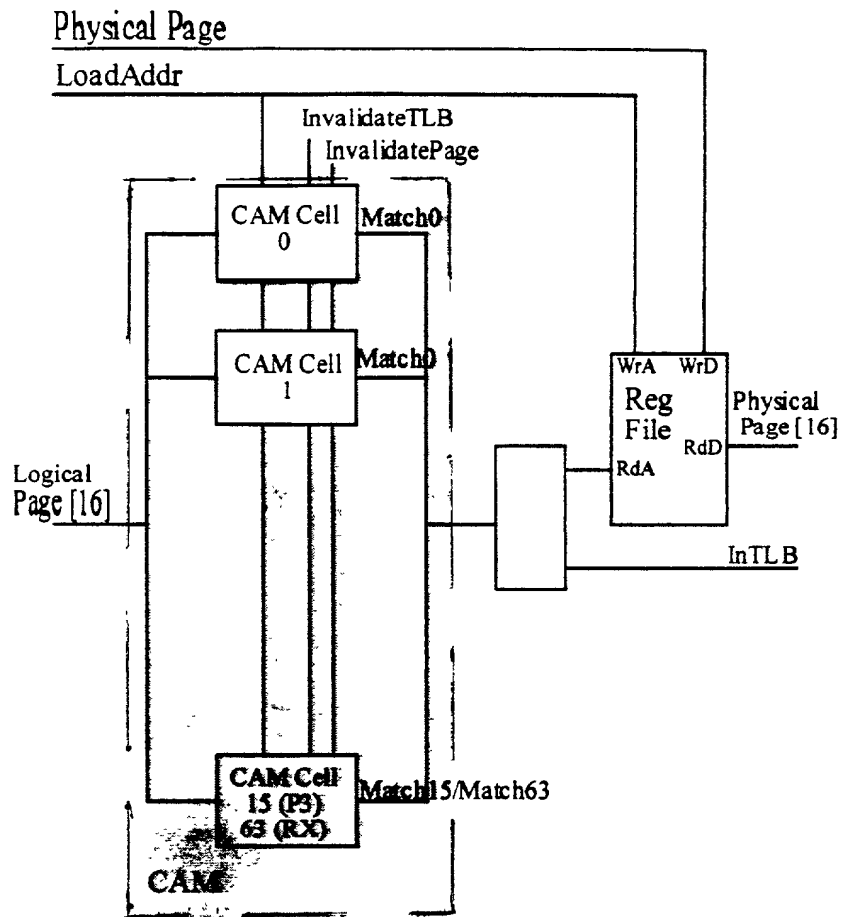


FIG. 14

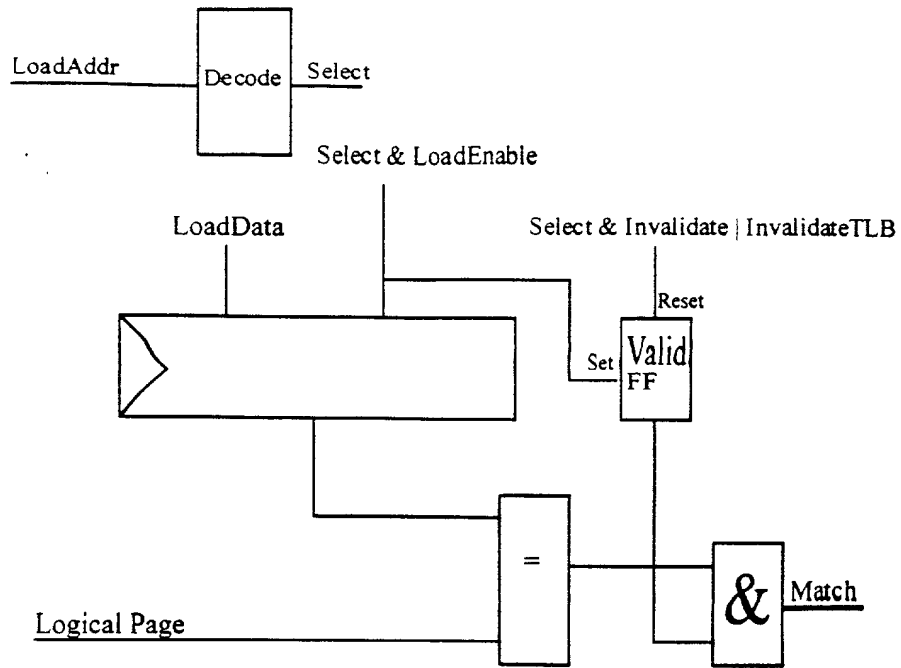


FIG. 15

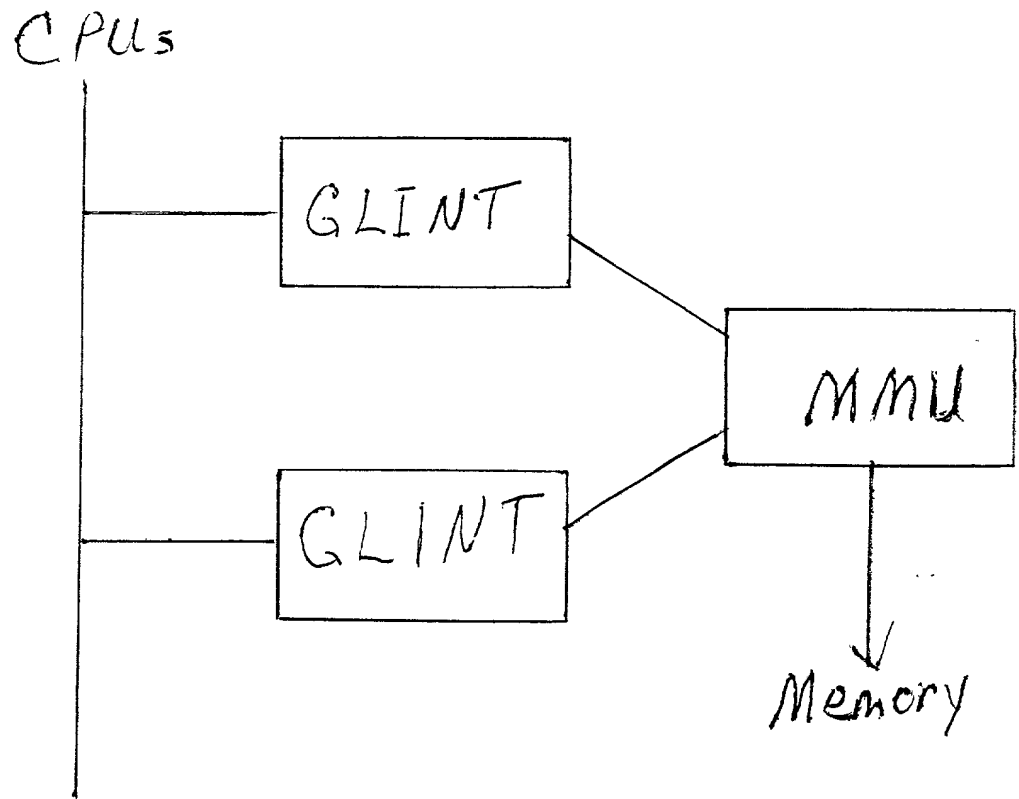


FIG. 16



## **DECLARATION AND POWER OF ATTORNEY**

As a below-named inventor, I hereby declare that:

My residence, post office address and citizenship are as stated below.

I believe that I am the original, first and sole inventor of the innovative subject matter described and claimed in the U.S. patent application identified as follows:

**Title: Graphic Memory Management With Invisible Hardware-Managed Page Faulting;**

Inventors: Dave Baldwin (full legal names listed below);

Attorney Docket No.: TD-155;

■ (if checked) which is attached hereto.

This application claims priority from parent 60/138,350 filed 06/09/99, 60/138,248 filed 06/09/99, and 60/143,822 filed 7/13/99 (MM/DD/YY).

I hereby state that I have reviewed and understand the contents of the above-identified U.S. patent application, INCLUDING THE CLAIMS.

I acknowledge the duty to disclose information which is material to the examination of this application in accordance with Title 37, Code of Federal Regulation, §1.56(a).

I do not know and do not believe that the claimed invention was ever known or used in the United States of America before my invention or discovery thereof.

I do not know and do not believe that the claimed invention was ever patented or described in any printed publication in any country before my invention or discovery thereof.

I do not know and do not believe that the claimed invention was ever patented or made the subject of an inventor's certificate issued prior to the date of this application in any country foreign to the United States of America on an application filed by me or my legal representatives or assigns.

I do not know and do not believe that the claimed invention was ever patented or described in any printed publication in any country more than one year prior to the filing date of this U.S. application.

I do not know and do not believe that the claimed invention was ever in public use or on sale in the United States of America more than one year prior to the filing date of this U.S. application.

I hereby appoint Robert Groover III, Reg.No.30,059, and Betty G. Formby, Reg.No.36,536 to prosecute this application and transact all business in the Patent and Trademark Office connected therewith, and also to file and prosecute any corresponding application in any foreign country.

I hereby direct that all correspondence and telephone calls be addressed to:

Robert Groover,  
17000 Preston Rd. #230, Dallas TX 75248;  
(972) 380-6333.

I hereby claim the benefit of priority, under 35 U.S.C. §119 or §120 as may be

applicable, of any application(s) for patent or inventor's certificate listed below:  
60/138,350 filed 06/09/99, 60/138,248 filed 06/09/99, and 60/143,822 filed 7/13/99  
(MM/DD/YY).

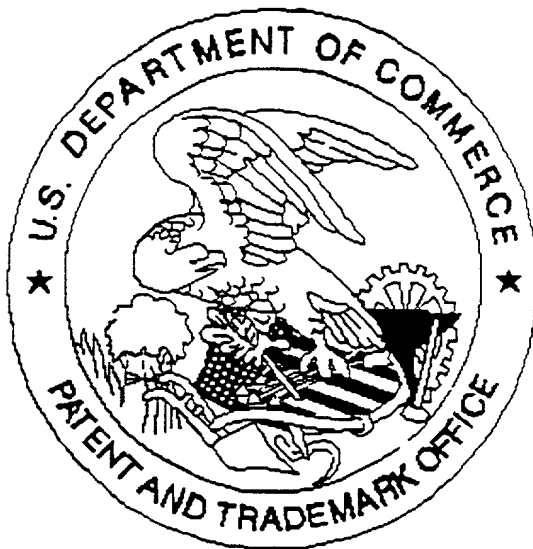
I hereby declare that all statements made of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements are made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and may also jeopardize the validity of the application or any patent issued thereon.

Inventor: David Robert Baldwin

Date: \_\_\_\_\_ Signature: \_\_\_\_\_

Residence and Mailing Address: Briarside, Gower Road, Weybridge, Surrey KT13 0H,  
UK      Citizenship: U.K.

United States Patent & Trademark Office  
Office of Initial Patent Examination -- Scanning Division



Application deficiencies were found during scanning:

☐ Page(s) \_\_\_\_\_ of \_\_\_\_\_ were not present  
for scanning. (Document title)

☐ Page(s) \_\_\_\_\_ of \_\_\_\_\_ were not present  
for scanning. (Document title)

☒ Scanned copy is best available.

*drawings*